AFIT/GCE/ENG/92D-08

Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms

THESIS

Laurence D. Merkle
Captain, USAF

AFIT/GCE/ENG/92D-08

AFIT/GCE/ENG/92D-08

Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Laurence D. Merkle, B.S.
Captain, USAF

December, 1992

Acknowledgements

Throughout this research effort, I have been the recipient of inestimable assistance from many sources. The value and clarity of the thesis benefited greatly from the sound and insightful advice of Lt Col William C Hobart, Jr, Maj Gregg H Gunsch, and Dr Henry Potoczny. The burden for any remaining defects remains mine. I am particularly indebted to Maj Gunsch for his knowledgeable AI lectures, and to Lt Col Hobart for his expertise in parallel architectures. My most heartfelt thanks go to my advisor, Dr Gary B Lamont, who set lofty goals I never imagined I could never meet, and somehow motivated me to meet them.

Insight has arrived from sources other than the AFIT faculty as well. Venkantarama Kommu led me to the conditional sharing strategy, while Kalyonmoy Deb has patiently answered endless e-mail questions about the messy genetic algorithm. Mr Richard L Norris, AFIT's dependable iPSC/2 system administrator, suffered untold system management requests and technical questions. Dr Ruth Pachter loaned me treasured books and spent long hours helping me learn about polypeptides and the conformational analysis problem.

This undertaking would not have been possible without the encouragement of my friends at Central Christian Church and the moral support of my new family, DeFro, Nancy, Steve, Doug, and especially Mary Lou. My friends Bill, Janet, Katherine, and Kelly were always there to revitalize me when my motivation dwindled. My parents, Doug and Carol, gave me a fascination for knowledge, taught me the importance of hard work, and gave me a loving environment in which to grow. Most important in these acknowledgements, and in my life, is my loving wife and best friend, Margaret. She more than patiently indulged my long hours and preoccupation with work.

Laurence D. Merkle

Table of Contents

List of Figures

List of Tables

Abstract

Genetic algorithms (GA) are highly parallelizable, robust semi-optimization algorithms of polynomial algorithmic time complexity. GAs are inspired by and modeled after the processes of natural selection. The most commonly implemented GAs are "simple" GAs, which use three standard genetic operators. Reproduction, crossover, and mutation operate on populations of strings, each of which represents a solution to a domain problem. Deceptive and GA-hard problems are provably difficult for simple GAs due to the presence of misleading building blocks and long building block defining lengths. Messy GAs (MGA) are designed specifically to overcome the limitations associated with deception and defining length. Reported MGAs have been applied to a number of functional optimization problems, and researchers have suggested application of the MGA to combinatoric optimization problems. This study extends AFIT's existing GA capabilities by generalizing the MGA to solve combinatoric optimization problems.

The performance of the generalized MGA is compared to that of AFIT's original MGA, the GENESIS simple GA, and the permutation version of GENESIS using three problems. In an application to a fully deceptive binary function optimization problem the generalized MGA consistently obtains better solutions than the simple GA. In an application to an NP-complete permutation problem, the generalized MGA again consistently obtains better solutions than the other three GAs. In an application to DeJong function f2, the generalized MGA obtains better solutions than the original MGA, but not as good as the simple GA.

AFIT's initial parallel implementation of the MGA obtained speedup by distributing members of the initial population to processors using an interleaving strategy. This study compares the solution quality and execution time obtained using interleaving and three other distribution strategies. Distribution strategy is not found to significantly affect solution quality. The indexed, modified indexed, and block distribution strategies all obtain "super-linear speedup" of the primordial phase, indicating that the efficiency of the sequential algorithm can be improved.

Population partitioning in parallel genetic algorithms requires design decisions concerning implementation of the selection and crossover operators. Experiments are performed comparing the solution quality, execution time, and convergence characteristics of three selection algorithms and three approximations to global crossover.

Generalization and Parallelization of Messy Genetic Algorithms

and

Communication in Parallel Genetic Algorithms

I. Introduction.

A large class of important problems cannot be solved to optimality within acceptable amounts of time using currently available methods. Advances in computer hardware design continue to extend the usefulness of these methods, but there is a limit. In order to fully address the problem, hardware advances must be accompanied by algorithmic advances. This thesis investigation explores the abilities of genetic algorithms to solve difficult and important problems for which no satisfactory methods have been identified.

This chapter describes the general issue motivating this thesis effort and introduces the basic concepts upon which it is founded. Limitations of current and foreseeable computers (Section 1.1), the notion of search (Section 1.3) the fundamentals of parallel computing (Section 1.2, and the general problem addressed (Section 1.4) are discussed. The remainder of the chapter presents the specific research objectives (Section 1.5), the assumptions regarding future advances in computing performance and readership requirements (Section 1.6), current knowledge (Section 1.8), and the general approach taken (Section 1.9). The organization of the remaining chapters is presented in Section 1.10.

1.1 Limitations of Computing Capabilities.

One difficulty with many optimization algorithms is that the execution time grows as an exponential function of the problem size. The limitations imposed by the so-called combinatoric explosion cannot be significantly lessened by future advances in computer hardware design, because even a thousandfold increase in hardware performance only slightly increases the size of problems which can be solved within practical time constraints. Figure 1 illustrates this concept.

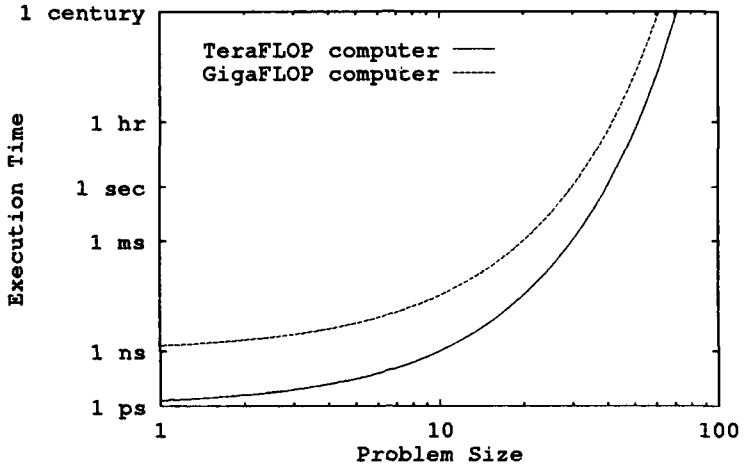

Figure 1. Theoretical Computation Times for an Exponential Complexity Algorithm on a GigaFLOP and a TeraFLOP Computer

One way to reduce the required execution time, and thereby increase practical problem sizes, is to accept near or approximately optimal solutions. Such a solution is often referred to as *semi-optimal.* Likewise, problems for which semi-optimal solutions are acceptable are referred to as semi-optimization problems. In most cases, it is possible to arrive at a semi-optimal solution in a much shorter amount of time. Figure 2 shows theoretical problem sizes for example exponential and polynomial time complexity problems on a TeraFLOP computer[1]. In order to fully exploit future hardware advances, future applications must exploit algorithms for which the execution time grows as a polynomial function of the problem size.

1.2 Parallel Computing.

Technology breakthroughs such as vacuum tubes, transistors, integrated circuits, and Very Large Scale Integration (VLSI) have had enormous impacts on computing performance. Optical computing promises to have an equivalent impact. Unfortunately, according to DeCegama, "computers are approaching a fundamental physical limit ...[and]

[1] A TeraFLOP computer is a computer capable of performing one trillion floating point operations per second. At the time of writing, no TeraFLOP computer exists.

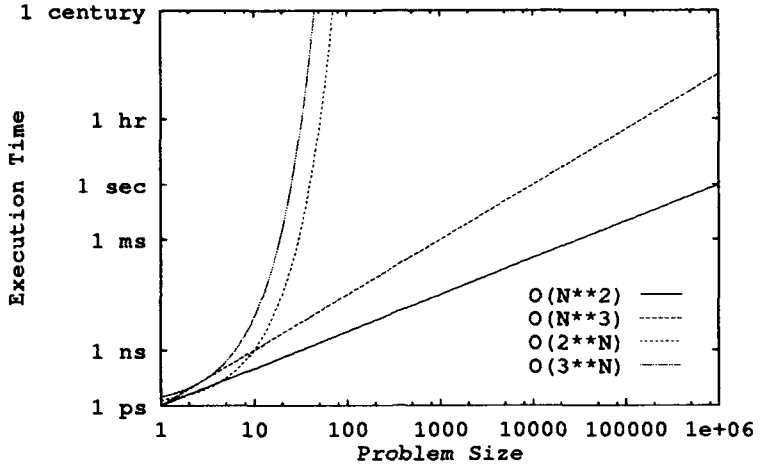

Figure 2. Theoretical Computation Times for Exponential and Polynomial Complexity Algorithms on a TeraFLOP Computer

sequential processors will reach the maximum speed physically possible in the near future" (12:2). Consequently, increased computing performance demands parallel computer architectures.

Several forms of parallelism exist which can lead to performance improvements when algorithms are implemented in parallel, including data parallelism, control parallelism, and trivial parallelism. These properties are discussed in Chapter II. The properties which can be most profitably exploited depend upon both the algorithm and/or application in question, and the target architecture.

A wide variety of parallel computer architectures have been designed and implemented. High level design options include shared/individual instruction and data streams, synchronous/asynchronous processing, ratios of individual processor speed to communication speed and processor memory size, local/global memory access, and processor interconnection topology. No single architecture has been shown to be clearly superior for all applications.

1.3 Search Algorithms.

Many problems can be posed as *search* problems, in which an algorithm examines the space of candidate solutions seeking points which satisfy domain specific criteria. One broad class of problems fitting this category is combinatoric optimization. An example of such a problem is the classic Traveling Salesman Problem (TSP), the objective of which is to construct a "tour" from a list of cities such that every city is visited exactly once and the total distance traveled is minimized. Many algorithms exist which solve these problems. Well known search algorithms which are often applied to combinatoric optimization problems include depth first, breadth first, best first, A*, branch-and-bound, hill climbing, and simulated annealing. Some are more efficient, some give solutions closer to the global optimum, and some are more widely applicable. The qualitative ability of an algorithm to perform well across a broad spectrum of problem domains is sometimes referred to as *robustness*, which Goldberg defines as "the balance between efficiency and efficacy." (21:1-10)

Another class of problems which can be described in terms of search is function optimization. For example, the problem of finding the roots of a function can be thought of as searching through the real numbers (the problem space) for points at which the value of the function is zero. Many numerical search algorithms exist which solve such problems, each of which has its own strengths and weaknesses. Some are more efficient than others, but are accurate only when applied to "well-behaved," continuous, or differentiable functions. Others are widely applicable or very accurate, but less efficient.

1.4 Problem.

Current and future challenges in computational science demand the application algorithms which are capable of fully exploiting the supercomputer architectures of the future. This implies polynomial complexity, highly parallelizable, robust algorithms.

Genetic algorithms (GA) solve semi-optimization problems, and exhibit all the above characteristics. GAs find applications in functional optimization, combinatoric optimiza-

tion, machine learning, and many other areas[2]. A recent variation of the GA, called the messy genetic algorithm (MGA) provides an even greater range of applicability.

A Genetic Algorithm Toolbox is being developed at AFIT to explore the capabilities of genetic algorithms. The toolbox has a sequential version implemented on the Sun 4 workstation, and a parallel version implemented on the Intel iPSC/2 and iPSC/860 Hypercubes. To date, functional optimization applications of both the standard and messy genetic algorithms have been parallelized (14, 52) (See Figure 3). The goal is a robust package capable of accurate and efficient performance on both functional and combinatoric optimization problems. This investigation enhances the capabilities of the Genetic Algorithm Toolbox by generalizing the messy genetic algorithm to solve combinatoric optimization problems. It also provides additional insight into effective and efficient application and parallelization of messy genetic algorithms.

1.5 Objectives.

The research objectives of this thesis are to:

1. Investigate the ability of messy genetic algorithms to solve difficult combinatoric and functional optimization problems to global optimality. Goldberg has stated that "messy genetic algorithms now appear capable of solving many difficult combinatorial optimization [problems] to global optimality in polynomial time or better"(24:417).
2. Investigate methods of implementing simple and messy genetic algorithms on parallel architectures. Execution time and solution quality of parallel messy genetic algorithms are compared using varying data distribution strategies. Execution time, solution quality, and convergence characteristics of parallel simple genetic algorithms are compared using varying communication strategies.

1.6 Assumptions.

Several key assumptions include:

[2]It should be pointed out that optimization is but one of many possible applications of GA, but it is the one of interest here.

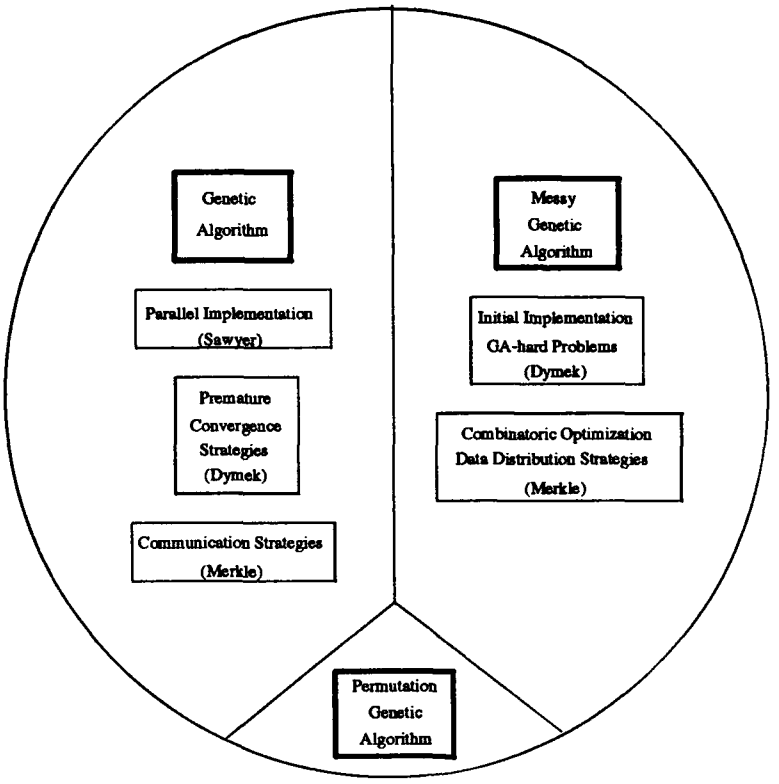

Figure 3. AFIT's Genetic Algorithm Toolkit (Current Status)

- There is an upper bound on the potential for increase in sequential computing performance.
- Future supercomputers exhibit ratios of inter-processor communication time to computation time similar to those of current supercomputers.
- The GENESIS simple and permutation genetic algorithms(27) operate correctly.
- AFIT's parallel messy genetic algorithm and parallel simple genetic algorithms(14) operate correctly.

Several additional assumptions are made regarding readership. Chapters III through VI assume familiarity with basic concepts of computer science and discrete math. In particular, it is assumed that the reader is comfortable with the concepts of algorithmic complexity and combinatorics. An understanding of genetic algorithms and parallel algorithms is helpful.

1.7 Scope.

With regard to the generalization of the messy genetic algorithm, it is shown that the generalized version is capable of solving

- a combinatoric optimization problem (the Traveling Salesman Problem),
- a difficult functional optimization problem (Rosenbrock's Saddle, also known as De-Jong function f2(13)), and
- the fully deceptive binary function optimization problem which has been solved by previously reported implementations of the messy genetic algorithm(14, 23, 24, 25).

The three problems are chosen specifically to demonstrate the messy genetic algorithm's ability to solve certain types of problems. The Traveling Salesman Problem (TSP) is a classic NP-complete problem(18). Any other NP-complete problem can be mapped to the TSP, and vice versa. Thus, the messy genetic algorithm's ability to obtain near optimal solutions to the TSP is indicative of its ability to obtain near optimal solutions for other NP-complete problems. Rosenbrock's Saddle is representative of a class of functions which are known to be difficult to optimize using standard gradient based techniques(13). It is also of practical importance in control systems(14). The fully deceptive binary function optimization problem is GA-Hard, i.e. provably difficult for simple genetic algorithms(20). Because the existence of GA-Hard problems is one of the primary arguments against the use of genetic algorithms, it is important to demonstrate the messy genetic algorithm's ability to solve this type of problem. The performance of the generalized messy genetic algorithm is compared experimentally to that of AFIT's original MGA implementation(14) and the simple genetic algorithm(27) on the basis of solution quality.

With regard to the parallelization of the messy genetic algorithm, four methods of data decomposition in the primordial phase are examined. One of the methods, which uses a standard interleaving approach, is the one used in AFIT's original parallel MGA implementation(14). Two others, which use modified interleaving, are strategies which Dymek discusses, but does not implement. The last strategy uses a standard block distribution approach. The strategies are chosen for their anticipated load balancing character-

istics. They are compared experimentally on the basis of solution quality and execution time for the fully deceptive binary function optimization problem.

Finally, a total of nine communication strategies for parallel genetic algorithms are examined, including all possible combinations of three selection strategies and three solution sharing strategies. The three selection strategies are all parallel implementations of an efficient and effective selection strategy commonly used in sequential genetic algorithms. Two of the solution sharing strategies are typical of parallel genetic algorithm implementations, while the third is an attempt to improve the performance of the second strategy. The strategies are compared theoretically on the basis of bias, spread, efficiency(3), and effects on schema growth. They are also compared experimentally on the basis of solution quality, execution time, and convergence characteristics.

1.8 Summary of Current Knowledge.

Current knowledge of messy genetic algorithms is limited to that established by Goldberg, et.al. (23, 24, 25), Deb (11), and Dymek (14). Published applications are limited to functional optimization problems with the exception of recent work by Deb (11). A number of successful combinatoric optimization applications of simple genetic algorithms have been reported.

A number of successful applications of parallel genetic algorithms have also been reported. Dymek(14) and Kommu(35) both investigate the effects of communication strategies on solution quality and execution time of parallel genetic algorithms. Dymek describes a parallel implementation of the messy genetic algorithm(14).

1.9 Approach.

AFIT's original implementation of the messy genetic algorithm(14) is extended to handle a wider class of problems, including combinatoric optimization problems such as the Traveling Salesman Problem (TSP). Structured analysis techniques(60) are applied in order to determine and document the requirements and specifications of the generalized messy genetic algorithm. The process descriptions are formally specified using the architecture

independent language UNITY(6) in order to facilitate parallelization. The performance of the generalized implementation on the functional optimization problem described by Dymek(14) is compared to the performance of the original implementation and the simple genetic algorithm. The three genetic algorithms' performance on the TSP and DeJong function f2(13) are also compared. In the case of the TSP, the performance of a version of the simple genetic algorithm which is specifically designed to solve permutation problems is also compared to the other three genetic algorithms' performance. All of the genetic algorithms are implemented on a Sun-4 in C.

The architecture independent description of the messy genetic algorithm is examined to identify reasonable methods of data distribution in the primordial phase. Four strategies are identified, and versions of AFIT's parallel messy genetic algorithm(14) using each are implemented on an iPSC/2 in C. The performance of the four versions in terms of solution quality and execution time in a series of executions is compared.

AFIT's parallel simple genetic algorithm(14, 53), which is implemented on an iPSC/2 in C, is modified to allow use of conditional solution sharing. The modified version allows three selection strategies and three solution sharing strategies. Dymek's premature convergence experiments are extended to compare the effects on solution quality, execution time, and convergence characteristics of the nine possible combinations of communication strategies.

1.10 Layout of the Thesis.

This chapter describes the motivation, objectives, and approach for this thesis. In Chapter II, the relevant genetic algorithm and parallel processing literature is summarized, and its impact on this work is assessed. Chapter III describes the requirements analysis and resulting design for a generalized version of the messy genetic algorithm. Experiments examining the performance of the generalized messy genetic algorithm are discussed in Chapter IV. Chapters V and VI describe experiments comparing several strategies for parallel implementation of the messy genetic algorithm and simple genetic algorithm, respectively. Conclusions regarding the experimental data are presented in Chapter VII. Finally, Chapter VIII offers recommendations for further research.

1.11 Summary.

Limitations of computing capabilities in the foreseeable future make necessary the study of widely applicable, polynomial complexity optimization algorithms. Simple and messy genetic algorithms are potential avenues for meeting the computational requirements of the future. This thesis describes a generalized implementation of the messy genetic algorithm, and experimentally compares its performance on several problems to the performance of other genetic algorithms. It also theoretically and experimentally compares various data decomposition and communication strategies for parallel genetic algorithms.

II. Literature Review and Requirements.

This chapter summarizes current literature related to the implementation of genetic algorithms (GA) on parallel computer architectures, beginning with the fundamental concepts of search (Section 2.1). Discussion of genetic algorithms begins with an overview of Holland's theories of adaptive systems, which led to the development of the first genetic algorithms (Section 2.2). The opertaion of a "simple" genetic algorithm is described(Section 2.3), followed by the basic theory explaining the effectiveness of genetic algorithms (Section 2.4).

Building upon this foundation, ordering operators for the simple genetic algorithm are discussed (Sections 2.5). Deceptive problems, which are the primary motivation for messy genetic algorithms, and the operation of MGAs are discussed (Section 2.6).

Discussion then turns to an overview of parallel computing (Section 2.7), including discussion of parallel architectures and parallel algorithms. This background is used to identify the key questions which arise in the parallel implementation of genetic algorithms (Section 2.8). The discussion includes a summary of how parallelism issues are addressed by other efforts in parallel genetic algorithms.

2.1 Search Algorithms.

As mentioned in Chapter I, many problems can be posed as search problems, in which an algorithm searches a space of candidate solutions for those points which optimize some domain specific function subject to certain constraints. Search is a versatile problem solving tool, because many practical problems can be easily represented as graphs(44:20-25). An example of such a problem is the classic Traveling Salesman Problem (TSP), the objective of which is to construct a "tour" from a list of cities such that every city is visited exactly once and the total distance traveled is minimized.

In such a representation, each *node* in the graph corresponds to an encoded portion of the problem, and each *edge* corresponds to an operation on the state of the problem[1].

[1] Pearl(44:33) correctly distinguishes between a directed *arc* and an undirected *edge*.

In the TSP example, each node represents a partial tour, while each edge represents the operation of adding a city to the tour.

Pearl lists node generation, exploration, and expansion as the key operations in graph searching(44:34). Node *generation* is the process of obtaining a new node from its parent. A node is said to be *explored* when at least one of its children is generated. Once all of its children are generated, a node is considered *expanded.* A search algorithm, then consists of an unambiguous specification of the order in which nodes are to be generated. Pearl distinguishes between *blind* search, in which the order in which nodes are generated is determined entirely by information obtained from the search process, and *informed* search, in which domain specific knowledge is applied to guide the search.

Brassard and Bratley identify five characteristics which are common to traditional search algorithms(4:79):

1. A set of candidates from which to construct a solution.
2. A set of candidates which have previously been considered.
3. A test for feasibility of a specific candidate.
4. A selection function which determines the next candidate to consider.
5. An objective function which assigns values to candidate solutions.

The process of solving an optimization problem consists of searching for a solution (node) or set of solutions for which the objective function is optimized, subject to the success of the feasibility function. Pearl classifies traditional search algorithms as hill climbing, uninformed systematic search, informed best-first search, specialized best-first search, and hybrid search(44:35-69). The distinctions are made primarily by the method which the selection function selects the next candidate for exploration.

Hill climbing strategies repeat the process of expanding the current node, selecting the most promising node from the children, and continuing the search from that node until a solution is found or the search fails. This strategy has a number of drawbacks which limit its applicability. First, depending upon the particular problem, hill climbing is not guaranteed to find a solution, or even to terminate. Second, again depending upon the

particular problem, even if hill climbing does find a solution, it is not guaranteed to be an optimal solution. In cases for which hill climbing is guaranteed to find an optimal solution, it is referred to as a "greedy" algorithm.

Important variations on hill climbing include simulated annealing(47) and Monte Carlo algorithms(4:262-267). Simulated annealing, which is so named because of the analogy to random atomic motions during annealing, allows the search to probabilistically move in a locally non-optimal direction. The probability of doing so is controlled by an "annealing schedule." Monte Carlo algorithms operate by repeatedly randomly selecting children nodes and comparing them to the current node. If the child node is better than the current node, it is adopted as the current node. Otherwise, the current node is retained.

In order to classify a search strategy as *systematic*, Pearl requires that it "not leave any stone unturned," and "not turn any stone more than once." (44:16) Uninformed systematic search strategies include depth-first, backtracking, and breadth-first. In each case, explored nodes are retained in memory in case the search reaches a dead end without finding a satisfactory solution.

In depth-first search(44:36), each node which is selected for exploration is fully expanded before any other node is explored. In choosing a node to explore, depth-first search gives preference to nodes at lower levels of the search tree. Backtracking(44:40) is very similar, except that nodes are not fully expanded when they are explored. Breadth-first search(44:42) is also very similar to depth-first search, except that preference is given to nodes at higher levels in the search tree. Breadth-first search has the significant advantage that for problems represented by locally finite graphs, and for which a solution exists, breadth-first search is guaranteed to find a solution and to terminate. On the other hand, in problems represented by graphs of bounded depth, depth-first search is also guaranteed to find a solution and to terminate, and it may do so much sooner than breadth-first search.

In informed systematic search, nodes are evaluated using a *heuristic* function, which uses domain specific knowledge to obtain a measure of merit for the node. The value of the heuristic function is used to determine which node to explore next. Commonly known informed systematic search algorithms include best-first search(44:48-56) and the

A* algorithm(44:64). Finally, Pearl discusses *hybrid* search strategies, in which portions of hill climbing, uninformed search, and informed search are combined.

An important consideration in the selection of a search algorithm for a particular problem is the execution time required to find the solution, and how the execution time is related to the problem size. Depending upon the problem to which they are applied, most of the search algorithms described above are of exponential algorithmic time complexity, i.e. their execution time grows exponentially with problem size. As discussed in Section 1.1, such algorithms quickly become intractable for practical problem sizes.

Another important consideration is whether or not the algorithm is guaranteed to find an optimal solution. Although optimal solutions are virtually always desirable, they often are not required. In many practical situations, it is more desirable to find a good solution in a reasonable amount of time than to find the optimal solution after it is no longer useful. Pearl discusses "near-optimization" and "approximate-optimization" tasks, both of which are subclasses of "semi-optimization" problems(44:15). Near-optimization tasks are those tasks for which the solution obtained must meet a specific acceptance criteria. Approximate-optimization tasks require only that the solution obtained be near-optimal "with sufficiently high probability."

2.2 Adaptive Systems.

Holland develops a theoretical framework for the analysis of complex adaptive systems. He cites as examples of such systems:

- Genetics/evolution
- Economics
- Control systems
- Physiological adaptation
- Game theory
- Machine Learning

His framework makes use of the following variables:

- the current environment, $E \in \mathcal{E}$, the set of possible environments
- the system's adaptive plan, $\tau \in \mathcal{T}$, the set of possible adaptive plans
- χ, the plan comparison criterion
- the set of structures currently in the system, $A \subset \mathcal{A}$, the set of possible structures
- the operators of the adaptive plan, $\omega_i \in \Omega$, the set of operators, and
- I, the set of possible inputs to the system, including μ, a measure of structure performance

Using this framework, a problem in adaptation is well posed once $\mathcal{T}, \mathcal{E}$, and χ have been specified. Likewise, an adaptive system is specified by $(\mathcal{A}, \Omega, I, \tau)$. Holland describes by way of his examples how complex adaptive systems can be described using the framework.

2.3 Simple Genetic Algorithms.

Goldberg defines genetic algorithms (GA) as "search algorithms based on the mechanics of natural selection and natural genetics" (21:1). More precisely, GA are algorithms for which the *inspiration* is derived primarily from theories of natural selection and genetics, via Holland's theories of complex adaptive systems(28). In keeping with mainstream literature, this thesis uses terms from the sciences of biology and genetics to refer to the software concepts which they have inspired. Goldberg describes in detail a particular genetic algorithm, which he calls the "simple" genetic algorithm (SGA). The vast majority of published genetic algorithm work has been based on the SGA, or slight variations thereof.

In the application of any GA to a particular problem, the first step is the design of an encoding scheme. An SGA encoding scheme is a one-to-one mapping from the problem's solution space to a fixed length binary string. GA strings are often likened to biological chromosomes. The individual elements of the string are called *features*, and are analogous to the genes of a chromosome. The values which an individual feature can assume are

referred to as *feature values*, which are analogous to the *alleles* of a gene. The set of all alleles is the *genic alphabet.*

In the case of a function optimization problem, the encoding scheme can be as simple as concatenating the binary representations of the independent variables. For example, Figure 4 shows an encoding scheme for a function of two variables in which each is encoded as a six bit binary string. Encoding schemes for combinatoric optimization problems are often more complicated. For example, Figure 5 shows a possible encoding scheme for the Traveling Salesman Problem (TSP) which assigns each city a unique number, then concatenates each city's number in the order in which they are to be visited.

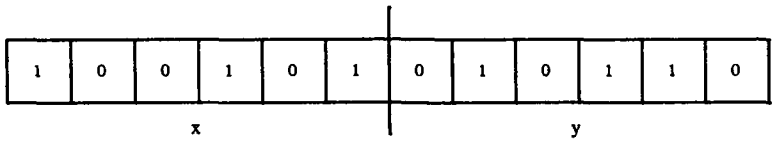

Figure 4. Encoding Scheme for Functional Optimization Problem $f(x, y), 0 \leq x, y < 64$

Figure 5. Encoding Scheme for 12 city Traveling Salesman Problem

Once the encoding scheme has been defined, a function which decodes candidate solutions and evaluates them relative to each other can be designed and implemented. The value returned by this function is called the solution's *fitness.* The fitness function corresponds to Brassard and Bratley's objective and feasibility functions, described in Section 2.1.

GAs begin execution by randomly generating a set of candidate solutions, referred to as the initial population. The number of solutions generated, the *population size*, is implementation dependent and can range from a few dozen to several thousand. It has been shown that the optimal population size for an application of the binary encoded SGA is a function of string length, degree of parallelization, and degree of convergence sought (19, 22). Once the initial population has been generated, GAs make repeated use of three basic operators, each of which is applied across the population: reproduction, crossover,

and mutation. These operators play the role of Brassard and Bratley's selection function[2]. One application of each operator completes an iteration, or "generation."

The reproduction operator is the means by which new generations are created from old ones. It randomly selects strings from the current generation to copy to the next generation, favoring those strings with higher fitnesses. It does not modify the strings. The most common implementation of the reproduction operator is stochastic sampling with replacement, more commonly known as "roulette wheel selection" (21:121). Many other methods have been proposed, including "Remainder Stochastic Sampling without Replacement," "Remainder Stochastic Independent Sampling," and "Stochastic Universal Sampling"(3).

Baker proposes three metrics by which to measure the performance of various selection algorithms(3). Bias measures the mean accuracy with which an algorithm allocates copies to solutions. Spread measures the maximum possible variation between the expected number of copies and actual number of copies allocated to a solution. Efficiency is the algorithmic complexity of the algorithm. Current theory of genetic algorithm behavior is based on the assumptions of zero bias and minimum spread. Thus, in order to achieve theoretically possible efficiency, an algorithm must exhibit these properties. Baker's metrics are discussed in more detail in Section 6.1.

The crossover operator is the primary mechanism by which new solutions are introduced to the search process. It randomly selects two "parent" strings from the current population, then randomly selects a crossover point within the length of the strings. Finally, it swaps the portions of the strings following the crossover point. This is illustrated in Figure 6.

The third basic operator, mutation, prevents stagnation in the search process. It is applied with low probability, typically less than 1% of the time. When it is applied, it randomly alters a randomly selected gene from the string. This is illustrated in Figure 7.

[2]It is important to note that, in the context of genetic algorithms, the term "selection" is often used in place of "reproduction." This is in contrast to Brassard and Bratley's "selection" function.

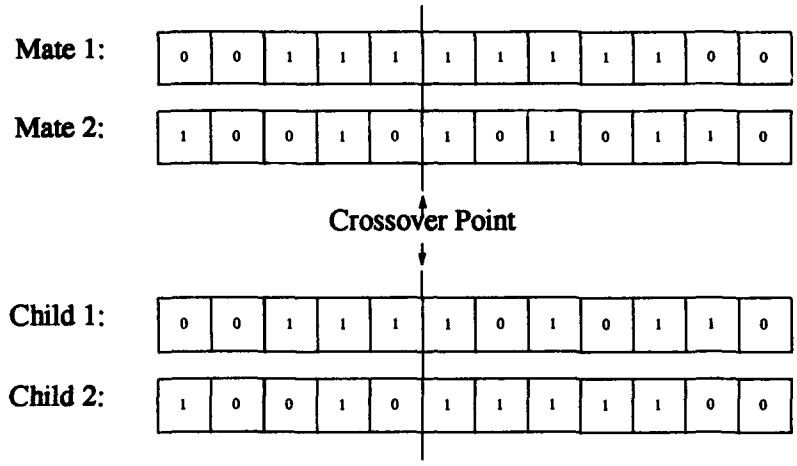

Figure 6. Crossover Operator

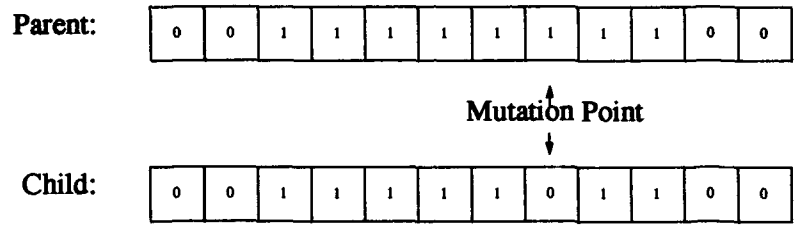

Figure 7. Mutation Operator

GAs repeat this process until some condition is met. A simple but common termination condition is that an arbitrary number of generations have been carried out.

2.4 The Schema Theorem and Deception.

Goldberg develops an estimate for the performance of the SGA(21:28-33). Theoretical analysis of GA performance makes extensive use of *schemata*, or similarity templates. Schemata are strings composed of characters taken from the genic alphabet, with the addition of the "don't care" character. A schema thereby describes a subset of the potential solutions. For example, the schema 1******* represents the set of all 8-bit strings which contain a 1 in the first position. Likewise, the schema 1******0 represents the set of all 8-bit strings which begin with a 1 and end with a 0.

Defining the average fitness of a string matching a schema H to be $f(H)$, the average population fitness to be $\bar{f}$, and the number of strings in a population at time t which match

the schema to be $m(H,t)$, the effect of the reproduction operator is

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}} \tag{1}$$

The defining length, $\delta(H)$, of a schema is the "distance" between the index of the first specified position and the index of the last specified position. For example, $\delta(1*****0*) = 7 - 1 = 6$, while $\delta(1*******) = 1 - 1 = 0$. Noting that crossover disrupts a schema only when the crossover point occurs within the defining length of the schema, the probability of survival under crossover for a schema in a string of length l is

$$p_s \geq 1 - p_c\frac{\delta(H)}{l-1} \tag{2}$$

where p_c is the probability of crossover and the inequality is used to reflect the fact that crossover may not actually disrupt the schema even when the crossover point is within the defining length.

The *order* of a schema H, which is denoted $o(H)$, is the number of specified positions in the schema. For example, $o(1*******) = 1$, while $o(11111111) = 8$. The probability of survival for the above schema under the mutation operator then can be estimated as

$$p_{ms} \approx 1 - o(H)p_m, p_m \ll 1 \tag{3}$$

where p_m is the probability of mutation. Combining these results and omitting negligible terms gives an estimate for the expected number of examples of a schema in the next generation:

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right] \tag{4}$$

This is referred to as the Fundamental Theorem of Genetic Algorithms, and can be interpreted as stating that "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations" (21:33). This result also goes by the name of the Schema Theorem.

The schema concept can be extended to apply to absolute and relative ordering problems. Following Kargupta(31), an absolute ordering schema defines a set of valid permutation strings. For example, the absolute o-schema ! 1 ! 5 ! ! represents the set of all permutation strings for which the second and fourth positions contain alleles 1 and 5, respectively. This o-schema is distinct from the standard schemata * 1 * 5 * * in that the former requires that the string represent a valid permutation, while the latter does not.

Following Goldberg(21), Kargupta uses $rs^{t}(H)$ to denote the set of all valid permutation strings in which the alleles specified in H occur in the specified order. For example, rs^{6}(1,5) represents all permutation strings of length 6 in which the allele 5 occurs after the allele 1.

2.5 Ordering Operators.

One implication of the Schema Theorem is that SGAs have difficulty solving problems in which important schemata (building blocks) have large defining lengths. One attempt to overcome this difficulty is the implementation of the inversion operator (21:166-170). The effect of the inversion operator, which operates on a single string, is to modify the ordering of the genes within a string. Specifically, when the operator is applied, two points within the string are randomly selected, and all genes within the string are inverted. The fitness of the string is unchanged, because the genes still map to the same domain parameters. Use of the inversion operator requires a more complex representation of the solution, since loci information must be represented explicitly.

Inversion complicates crossover, because two arbitrary strings are not guaranteed to share the same ordering. In fact, they are very unlikely to. A number of approaches have been taken to resolve this difficulty(21). Goldberg reports(21:169) that Frantz(17) investigated four mating rules in an attempt to prevent crossover between incompatible strings. "Strict homology" requires that mates share the same ordering (Goldberg credits this strategy to Bagley(2:168)). "Viability mating" allows strings with different orderings to mate, but their offspring is only retained if it is a valid, completely specified solution. "Any-pattern mating" randomly selects one of the two strings' orderings as the "prime"

ordering, and maps the other string to match the prime ordering prior to mating. Finally, "best-pattern mating" chooses the ordering of the more fit string as the prime ordering.

The motivation for the inversion operator is to allow the GA to search for good problem representations at the same time that it searches for good solutions. Unfortunately, it has never produced good results due to its unary nature(26). Davis(9), Goldberg and Lingle(26), and Smith(55) propose three other ordering operators which combine the effects of crossover and inversion. Partially Matched Crossover (PMX), Ordered Crossover (OX), and Cycle Crossover (CX) all combine the orderings of two solutions in such a way as two produce valid offspring. Each tends to preserve different properties of the original orderings. PMX tends to preserve absolute ordering, OX tends to preserve relative ordering, and CX tends to preserve cycles. Each has been shown to produce better results than simple inversion.

2.6 Deception and Messy Genetic Algorithms.

Another implication of the schema theorem is that problems for which short, low-order, above-average schemata do not combine to form globally optimal solutions prove difficult for SGA. An example of such a problem is shown in Table 1. The fitness gradient is such that the SGA tends toward solutions with fewer 1's at the specified positions, as shown in Figures 8 and 9(23:510). Thus, it may never find the global optimum which has 1's at all of the specified positions. Such problems are sometimes referred to as *deceptive*. Deceptive problems in which the deceptive building blocks have long defining lengths are called *GA-hard*. The function is *order-3 deceptive* because it is deceptive on a subproblem which is defined on 3 positions.

Goldberg designed messy genetic algorithms (MGA) specifically to solve deceptive problems(23, 24, 25). MGAs consist of a *primordial* and a *juxtapositional* phase(23). The MGA's ability to solve deceptive problems stems from the use of *partially enumerative initialization*, in which the initial population consists of all possible partial solutions of a specified length. It is created by first enumerating *building blocks*, and then distributing each building block across the genes in every possible combination. The building blocks are an exhaustive list of allele combinations of length equal to the *block size*, or suspected

Alleles	Average Fitness
0**0**0*********	28
0**0**1*********	26
0**1**0*********	22
0**1**1*********	0
1**0**0*********	14
1**0**1*********	0
1**1**0*********	0
1**1**1*********	30

Table 1. Order 3 Fully Deceptive Function

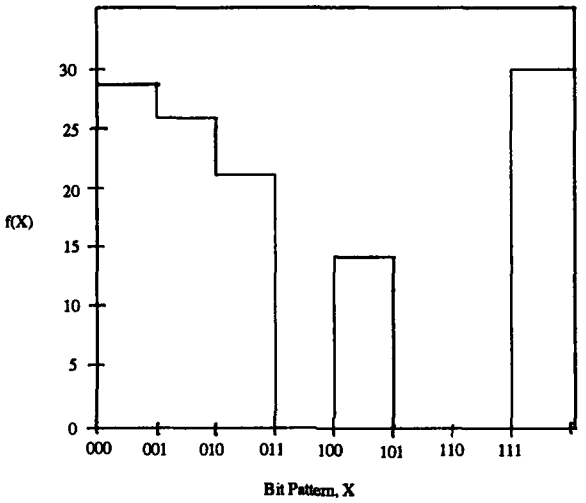

Figure 8. Hamming Graph of Deceptive Sub-function

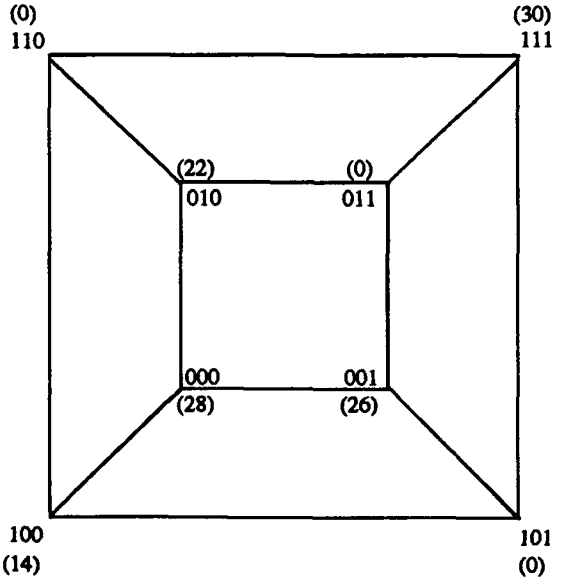

Figure 9. Fitness Gradient of Deceptive Sub-function

nonlinearity, of the domain problem. If the block size is greater than or equal to the level of deception present in the problem, partially enumerative initialization guarantees that all building blocks necessary to form the globally optimal solution will be represented in the initial population.

The use of partially enumerative initialization motivates several other differences between simple genetic algorithms and MGAs. First, MGAs must be capable of conducting meaningful competition between partially specified solutions. The MGA finds a locally optimal solution to the domain problem, called the *competitive template*, which it uses to "fill in the gaps" in under-specified solutions to allow their evaluation.

Another difference between simple genetic algorithms and MGAs is the size of the initial population. In an application of the MGA, there are k^C building blocks, and $\binom{\ell}{k}$ combinations of k genes, where k is the block size, C is the cardinality, and ℓ is the string length. Thus, the initial population contains

$$n = k^C \binom{\ell}{k} \tag{5}$$

solutions, which is significantly more than in a typical simple genetic algorithm application. The MGA *enriches* the initial population through the use of *tournament selection*, and periodically reduces the population size during selection. Tournament selection enriches the population by increasing the proportion of building blocks which lead to improvements to the solution represented by the competitive template.

The juxtapositional phase is similar to a simple genetic algorithm, with the main difference being that the MGA must handle variable length strings. For this reason, the crossover operator is replaced by a *cut and splice* operator. The behavior of the cut and splice operator is shown in Figure 10(14:103).

Recently, Kargupta has examined deception in the context of ordering problems(31). As standard deception[3] is related to schemata, ordering deception is related to ordering

[3]Because deception has only recently been examined in the context of ordering problems, there does not yet exist a widely accepted term to refer to "standard" deception while excluding ordering deception.

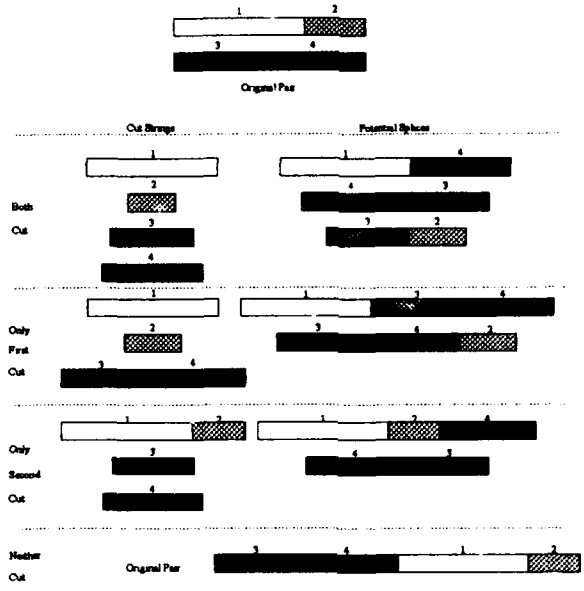

Figure 10. Cut and Splice Operations

schemata. Kargupta presents an order-4 fully deceptive absolute ordering problem and an order-4 fully deceptive relative ordering problem, and reports that Sikora(54) has demonstrated that no order-3 fully deceptive ordering problem exists. Prior to this thesis, the application of MGA to permutation problems had been suggested, but not addressed.

2.7 Parallel Architectures and Algorithms.

This section discusses two distinct but interdependent aspects of parallel computing. Section 2.7.1 considers issues related to the design and implementation of parallel computer architectures. Section 2.7.2 examines the design and implementation of algorithms which exploit application parallelism.

2.7.1 Parallel Architectures. The vast majority of computer architectures in common use are based on the organization proposed by von Neumann in the 1940s, in which a single memory area is used to store both instructions and data. Such architectures are referred to as von Neumann-based. Von Neumann-based parallel processing systems can be categorized as Multiple Instruction Multiple Data (MIMD), Single Instruction Multiple

For simplicity, the old term "deception" is hereafter used to refer to "standard deception" and references to ordering deception are made explicit.

Data (SIMD), Multiple Instruction Single Data (MISD), or Single Instruction Single Data (SISD). A special case of the MIMD category is the Single Program Multiple Data (SPMD) paradigm. Other architectures exist, but their use is primarily limited to research. The majority of commercially available parallel architectures are either SIMD or MIMD.

During any given instruction cycle, all of the processors of a SIMD architecture execute the same instruction, using different data. In order for the instructions to be applicable to the data on all the processors, they must be more general and therefore less powerful. As a result, the individual processors have small instruction sets, making them relatively inexpensive, so that it is cost effective for SIMD architectures to include large numbers of processors. SIMD architectures with 64,000 processors are fairly common. The tradeoff is that for a fixed amount of memory, there is less memory per processor.

In contrast, the processors of a MIMD architecture act independently, and can take advantage of more powerful instructions. Each processor is more expensive, so that MIMD architectures are typically implemented with fewer processors than SIMD architectures. This means that each processor can be allocated more memory.

A single processor and its allocated memory are together called a *node*. The relative computational power of each node in a parallel architecture is often referred to as the *granularity* of the architecture. Most SIMD architectures are categorized as *fine grained* because they have a large number of nodes, each of which has a simple processor with a small amount of memory. In contrast, most MIMD architectures are categorized as *coarse grained* because they have a relatively small number of nodes, each with a powerful processor and significant memory.

Some architectures allow processors to access memory allocated to other processors, or simply allow all the processors to access a single global memory. Such architectures are referred to as *shared memory* architectures. Processors within such architectures can communicate data by storing it in memory which is accessible to the receiving processor. Most SIMD architectures are in this category. Most MIMD architectures, on the other hand, are *distributed memory*, meaning that processors cannot access memory allocated to other processors. These architectures are also referred to as *message passing*, because the

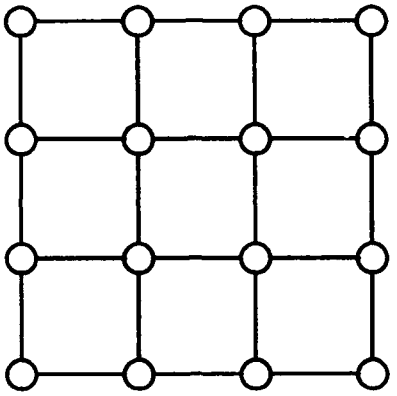

Figure 11. 4 x 4 Mesh Interconnection Network

processors communicate via communication links. This type of communication is generally very slow relative to other processor activities.

Parallel architectures can also be categorized according to their *interconnection topology*, or *network*, which defines the other processors to which each processor can communicate data. A common interconnection topology for SIMD architectures is a *2-D mesh*, in which the processors are arranged, logically if not physically, in a two dimensional array. A 4 x 4 mesh is shown in Figure 11. Mesh interconnection networks allow each processor to communicate data to each of the four processors at its sides. Well known examples of such systems are the Connection Machine, which is manufactured by Thinking Machines, Inc., and the Paragon, which is manufactured by Intel.

A common interconnection topology for MIMD architectures is a *hypercube*. Hypercube architectures have a *dimension*, N, and have 2^N processors. A hypercube of dimension 4 is shown in Figure 12. Each processor is directly connected to, and can communicate data to N other processors in a single step. Any processor can communicate data to any other processor in no more than N steps. One of many examples of a commercially available hypercube architecture is Intel Corporation's parallel supercomputer, the iPSC/i860.

2.7.2 Parallel Algorithms. Software development for parallel architectures is fundamentally different than for sequential architectures (6). The primary question in developing parallel software is whether to design parallel algorithms and implement them directly, or to implement sequential algorithms and then parallelize them. For most cases

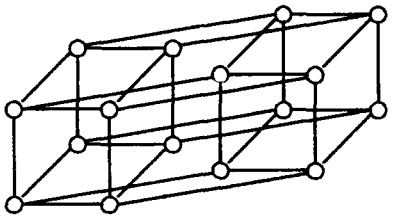

Figure 12. Dimension 4 Hypercube Interconnection Network

in which there is no sequential software predecessor, and even in some cases where there is such a predecessor, the first approach will likely result in better performance.

Several algorithm properties can lead to performance improvements when the algorithms are implemented in parallel. The two most common such properties are data parallelism and control parallelism(38). The former describes a situation in which an algorithm processes multiple data items in the same way, and the actions taken for any particular data item do not depend on the results of processing other data items. The latter is present when two distinct operations on the same data do not depend on each other. Another form of parallelism, sometimes referred to as "trivial" parallelism, is present when two separate activities, which share neither control nor data, can be executed simultaneously.

Chandy and Misra propose an architecture independent method for the description of an algorithm(6). A UNITY (Unbounded Nondeterministic Iterative Transformations) program describes the requirements for a process. It does not specify the order of operations or the mapping of operations to processors. Thus, a UNITY program may be mapped to any architecture, whether it be sequential, asynchronous shared-memory, or distributed memory. The description of a mapping describes how the UNITY program is executed on the target architecture. Mappings for particular classes of architectures exhibit common characteristics. The target architecture in this study is a hypercube, which is a distributed memory (DM) system. Chandy and Misra describe DM systems formally as consisting of a fixed set of processors, a fixed set of communication channels, and a memory for each processor(6:83). As such, a mapping to such an architecture must

- allocate each statement in the program to a processor;
- allocate each variable to either a memory or a channel;

- specify the control flow for each processor;
- allocate at most one variable, which is of type sequence, to each communication channel;
- be such that a variable which is allocated to a channel is referenced in statements which are allocated to exactly two processors.

Furthermore, the statements allocated to one of the processors which reference a channel variable may only modify the variable by appending an item to the sequence. They may only do so when the sequence is of length less than a constant buffer size. Finally, the statements allocated to the other processor which references the channel variable may only modify the variable by removing the first item in the sequence. They may only do so when the length of the sequence is greater than zero.

2.8 Parallelism in GA.

Genetic algorithms possess both data parallelism and control parallelism. The data parallelism is exhibited in the application of identical genetic operators to multiple data. In the case of the reproduction and crossover operators, the parallelism exists at the string level. In the case of the mutation operator, it exists at the gene level. The control parallelism present consists of the ability to overlap execution of operators. Control parallelism exists only within a single generation of a simple GA due to the requirement for calculation of average population fitness prior to application of the reproduction operator. Thus, in order to significantly increase the computational power which can be applied to genetic algorithms, a parallel implementation must seek to exploit the data parallelism.

In an implementation of a PGA based on data decomposition, the population is distributed amongst the various processors, each of which performs selection, crossover, and mutation, as shown in Figure 13(14:28). The portion of the population distributed to a given processor is referred to as a *subpopulation.* Key design decisions are how, when, and what information should be communicated between processors.

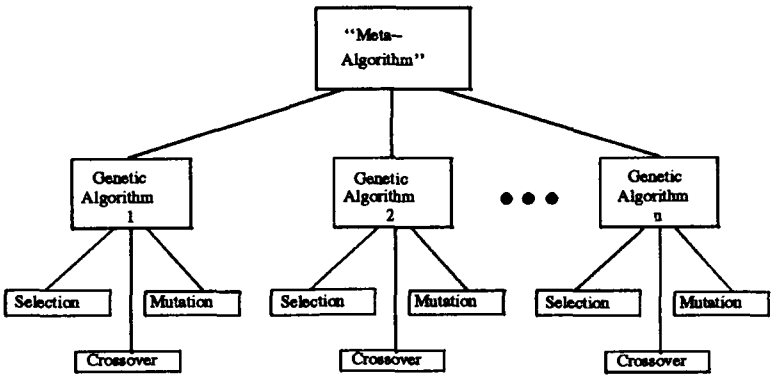

Figure 13. Typical Hypercube "Decomposition" of Genetic Algorithm

2.8.1 Hypercube Implementations. Global selection requires a global fitness calculation. In most reported hypercube implementations of genetic algorithms, the individual nodes compute subpopulation average fitnesses, which they communicate to an arbitrary node. This node then computes and communicates the global average fitness, which each node uses to perform selection locally. Some implementations perform this operation every generation (8, 7, 45, 30, 46), while others communicate only after a number of generations (58), performing purely local selection in between. It has not been shown that this approach either maintains or disrupts the Schema Theorem, although Pettey shows that the number of trials is bounded above exponentially(46).

Global selection also requires either frequent solution migration to maintain equal subpopulation sizes or relaxation of certain constraints, such as fixed population size(14). The latter affects the allocation of copies to solutions, possibly leading to premature convergence.

Because global selection results in large communication requirements, some hypercube implementations implement local selection(14). Current theory does not address the effectiveness of genetic algorithms using local selection.

2.8.2 Fine Grained Implementations. Implementations of PGA on fine grained SIMD architectures, such as the Connection Machine, assign much smaller subpopulations to each processor than are typically used in hypercube implementations(51) (49) (40) (57) (37) (41). In some cases, each subpopulation consists of a single string. Reproduction and

crossover are performed in "neighborhoods" of processors. A fairly common definition of a processor's neighborhood is the set of processors within a specified "distance" of that processor in a 2-D (wraparound) mesh or ladder[4].

Neighboring processors tend to converge on similar solutions, so that in mesh implementations, the effectiveness of the crossover operator is limited to those processors at the borders of "colonies." As a result, trial allocation is bounded by a quadratic function as opposed to an exponential function(57), thus reducing the effectiveness of the genetic algorithm.

There are various methods for getting around this problem, including global distribution of the globally best solution, "scattering"(37) and hill climbing(40). All have been shown empirically, though not theoretically, to be effective at finding near optimal solutions.

2.9 Kruskal-Wallis H Test.

This thesis presents the results of a number of experiments in which various implementations of genetic algorithms are compared. In order to draw meaningful conclusions from the results of those experiments, statistical hypothesis testing is required(1:483). Many statistical tests are applicable only when the data may be assumed to be normally distributed. Such tests are not appropriate in most of the experiments in this thesis. The populations in these experiments cannot be assumed to be normally distributed because there is a known bound on the experimental data. For example, there is a lower bound on the solution quality for the Traveling Salesman Problem experiments, corresponding to the tour length of the optimal solution. The existence of a bound on the experimental data is inconsistent with the assumption of a normal distribution. Furthermore, in most cases the bounds are near the observed means of the experimental data, so that the errors introduced by the assumption of a normal distribution are likely to be quite large.

The Kruskal-Wallis H Test determines "whether or not the means from k independent samples are equal when the populations [cannot be assumed to be] normal"(1:544). The

[4]Muhlenbein uses a "ladder" architecture, which is essentially a mesh of width two(41).

Suppose we have k indpependent samples from k populations. We wish to test the null hypothesis

H_0 : the samples are from identical populations

against the alternative hypothesis

H_1 : the populations are not identical

at the α level of significance.

1. Compute h. *Calculate*

$$h = \frac{12}{n(n+1)} \sum_{i=1} k \frac{R_i^2}{n_i} - 3(n+1)$$

2. Accept or reject H_0. *If $h > \chi^2_{k-1,\alpha}$, reject H_0; otherwise accept H_0.*

Figure 14. Kruskal-Wallis H Test Algorithm
(1)

algorithm for the Kruskal-Wallis test is given at Figure 14, in which n is the total number of observations, k is the number of samples, and R_i is the rank of observation i within the popluation. The Kruskal-Wallis H Test is used throughout.

III. Generalization of the Messy Genetic Algorithm.

The messy genetic algorithm's ability to solve deceptive problems of the type discussed previously (Section 2.6) has been demonstrated, and is central to claims of its robustness in the context of functional optimization problems(23, 24, 25). The MGA's ability to solve deceptive ordering problems(31) has not been addressed, and therefore robustness claims do not currently include applicability to permutation type combinatoric optimization problems. This chapter discusses the generalization of the MGA to permutation problems as well as functional optimization problems. It also examines the relative performance of the generalized MGA compared to AFIT's original MGA implementation(14) and the GENESIS simple genetic algorithm(27).

Dymek(14) uses structured analysis techniques(60) to develop requirements, specifications, and a high level design of the messy genetic algorithm based on Goldberg's descriptions(23, 24, 25). As an alternative to structured analysis, Rumbaugh describes object oriented analysis and design techniques(50). In order to obtain the maximum benefit from reuse, Dymek's structured analysis is revised for the generalized MGA. The requirements, which are presented in the form of a hierarchy of data flow diagrams, are modified to reflect changed and additional requirements for the generalized MGA implementation (Section 3.1). The specifications, which are presented as UNITY process descriptions are modified accordingly (Section 3.2), as is the high level design, which is presented as a structure chart (Section 3.3).

3.1 Requirements Analysis.

3.1.1 Problem Statement. The generalized messy genetic algorithm is a stochastic semi-optimization algorithm applicable to both functional and combinatoric optimization problems. It accepts as input at run time the parameters shown in Table 2. The user must define the function to be optimized in an evaluation, or "fitness," function. The user must also specify the optimization criteria, e.g. whether the function is to be maximized or minimized. Specifics regarding the form of the fitness function and the optimization

Parameter	Syntactic Meaning
String Length (l)	number of genes in each fully specified string
Block Size (k)	estimated nonlinearity, or level of deception, of the problem
Reduction Factor	factor by which population size is reduced during reducing tournaments
Reduction Interval	number of generations between population reducing tournaments
Number of Reductions	number of primordial phase population reducing tournaments
Shuffle Factor	number of strings examined in a search for a compatible opponent
Bitwise Cut Probability (p_κ)	probability per bit that a string will undergo a cut
Mutation Probability (p_m)	probability that a bit will undergo a mutation
Total Generations	sum of primordial phase and juxtapositional phase generations
Overflow Factor	maximum ratio by which actual string length may exceed l

Table 2. Messy Genetic Algorithm Parameters

criteria are left to the low level design (See Section 3.4). At completion the MGA outputs the best solution found, as well as the fitness of the solution.

3.1.2 Context Diagram. The context diagram for the generalized MGA, shown in Figure 15, reflects the generalization of Dymek's implementation to both functional and combinatoric optimization problems(14:67). There are two differences between Figure 15 and the context diagram for Dymek's implementation. The first difference is the substitution of the `initialization_function` flow for the `genic_alphabet` flow. This change is motivated by the tendency for combinatoric optimization problems to require genic alphabets of large cardinality. For example, the cardinality of the TSP is equal to the problem size(22). In contrast, functional optimization problems are often best represented using a binary genic alphabet(22). For example, a functional optimization problem in which each of two independent variables ranges from -2.048 to 2.047 would be very naturally represented by a 24 bit binary string. As presented by Dymek(14), the string is divided into

two segments of 12 bits, each of which is interpreted as a signed integer and then divided by 1000. Each segment thus has a range from $\frac{-2^{12}}{1000} = -2.048$ to $\frac{2^{12}-1}{1000} = 2.047$

Run time specification of the genic alphabet requires explicit specification of a character to represent each possible value of a gene. This is impractical if not impossible for large combinatoric optimization problems, for the simple reason that there is a limited number of ASCII characters. Therefore, the generalized MGA does not explicitly accept the genic alphabet as run time input. Rather, it requires the definition of a *domain initialization function*, which must generate the genic alphabet. The generalized MGA uses the cardinality of the genic alphabet to generate a genic alphabet. The domain initialization function can also perform any necessary domain specific initializations, such as accepting the distance matrix defining an instance of the TSP.

The second difference between Figure 15 and Dymek's context diagram is the addition of the `overlay_function` flow. The addition is motivated by the observation that for many combinatoric optimization problems, not all combinations of alleles constitute valid solutions. This complicates the evaluation of partially specified solutions, because a simple overlay operation of the partial solution on the competitive template may not result in a completely specified solution. Referring again to the TSP example, overlaying a partial tour on a completely specified tour is very likely to result in a tour which visits some cities twice and other cities not at all. It also may result in a tour in which each city is listed once, but the tour is actually composed of multiple cycles. Therefore, the generalized MGA implementation requires the definition of a third user defined function. The *domain overlay function* must accept a partially specified solution and a competitive template and must return a completely specified solution based primarily on the partially specified solution. There is no requirement concerning the optimality of the solution returned by the overlay function.

3.2 Specifications.

The Level 1 data-flow diagram (DFD) for the generalized MGA, shown at Figure 16, differs from Dymek's in two respects in order to reflect the additional requirements discussed previously and to explicitly acknowledge a previously existing requirement(14:67).

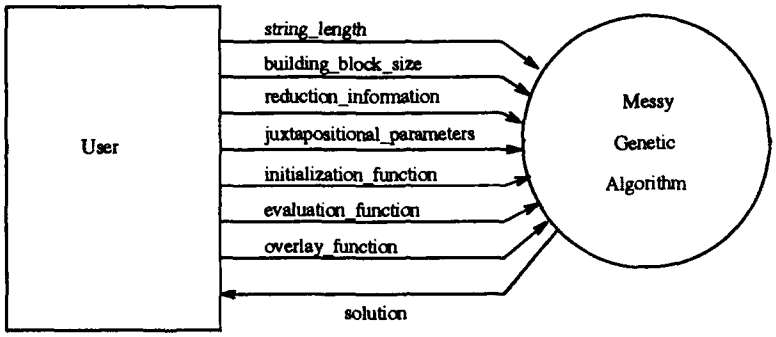

Figure 15. Context Diagram for a Messy Genetic Algorithm

First, it documents the domain initialization and domain overlay function requirements. Second, it indicates the requirement for the availability of the competitive template information in the juxtapositional phase. The fundamental requirements remain unchanged. The MGA initializes a population, enriches that population, and applies genetic operators to the population to generate a semi-optimal solution. All of the data elements used in the requirements analysis are defined in the data dictionary (Table 3).

The Level 2 DFD for the initialization phase is shown in Figure 17. It depicts the three key processes of the initialization phase: the generation of the competitive template, the creation of the building blocks, and the distribution of the building blocks to create the initial population. Separate generation and distribution of the building blocks would ideally be treated as an implementation decision, and not specified as a requirement. It is included here to retain agreement with Goldberg's description of the partially enumerative initialization process(23) and Dymek's associated implementation(14). Any design which generates the same initial population satisfies the problem statement, regardless of whether or not the building blocks are ever explicitly generated.

The Level 2 DFD for the primordial phase, which is shown at Figure 18, differs from Dymek's in several respects. Dymek decomposes the **Initialize Population** process into **Reproduce Population** and **Decrease Population**, which does not explicitly acknowledge the presence of tournament selection in the primordial phase(14:68). Figure 18 decomposes the **Initialize Population** process into into the **Select Strings** and **Conduct Tournament** processes, which capture the important data transformation ac-

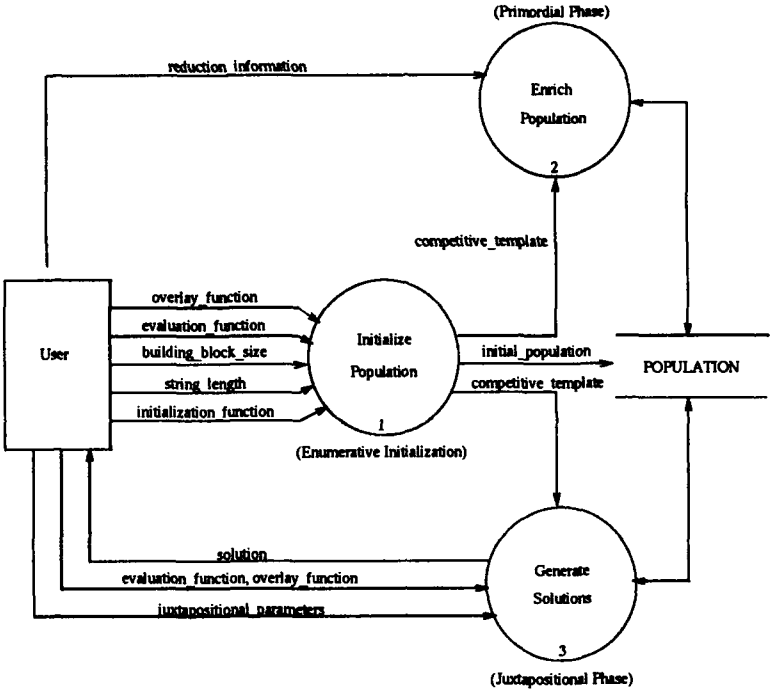

Figure 16. Level 1 Data Flow Diagram for a Messy Genetic Algorithm

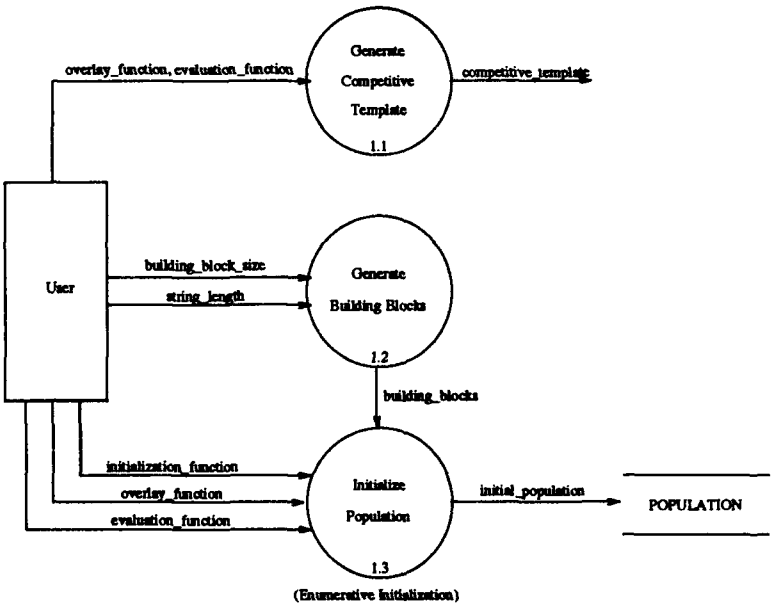

Figure 17. Level 2 Data Flow Diagram for Initialization Phase

Data Element Name	*Definition*
allele	*the value of a gene* $\in$ genic_alphabet
building_block	*partial solution of length building_block_size* string
building_block_size	*highest order suspected nonlinearity* *range: 1–string_length*
competitive_template	*locally optimal string* string
cut_probability	*the likelihood a string will undergo a cut* *range: 0.0–1.0/(string_length)*
cut_string	*string which has just undergone a cut* string
evaluation_function	*user-supplied function which evaluates how well the string solves the problem* f(string) $\rightarrow$ number
gene	*atomic unit of a string* allele + locus
initial_population	*the starting population*
initialization_function	*user-supplied function which generates genic alphabet and initializes evaluation_function*
juxtapositional_parameters	*input parameters for the juxtapositional phase* total_generations + cut_probability + splice_probability
genic_alphabet	*the values which may be assigned to an allele*
locus	*position of a gene within a string* *range: 1–string_length*
overlay_function	*user-supplied function which combines partially specified solution with competitive template to form completely specified solution*
POPULATION	*all strings currently existing in the MGA* 2{string}
reduction_information	*parameters specifying the reduction strategy* reduction_rate + reduction_interval + reduction_total
reduction_interval	*number of tournaments between reductions* integer
reduction_rate	*ratio by which population is reduced* *range: 0.0–1.0*
reduction_total	*total number of reductions* integer
solution	*best solution found by the MGA* string
splice_probability	*probability two cut_strings will be concatenated* *range: 0.0–1.0*
string	*solution encoded in the genic alphabet* 1{allele + locus} = 1{gene}
string_length	*length of string which fully specifies a solution*
string_pair	*two randomly chosen strings* 2{string}
total_generations	*number of generations in juxtapositional phase*

Table 3. Data Dictionary for Messy Genetic Algorithm

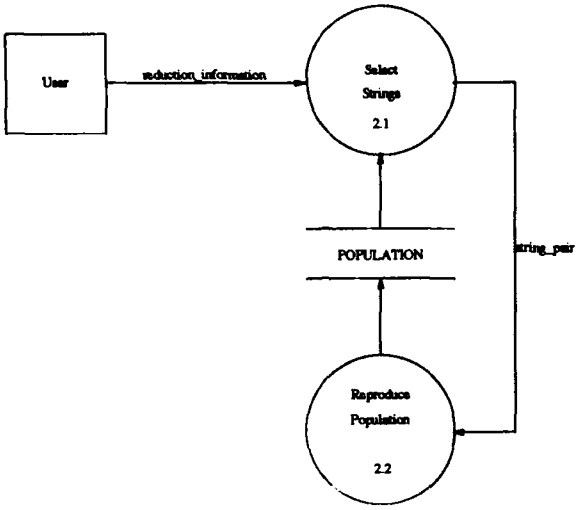

Figure 18. Level 2 Data Flow Diagram for Primordial Phase

tivities of the primordial phase. It also explicitly acknowledges the requirement for the **reduction_information** flow to the **Select Strings** process.

The Level 2 DFD for the juxtapositional phase is given at Figure 19. It differs from Dymek's in several respects, again in order to reflect the additional requirements discussed above, acknowledge previously existing requirements, and model the pertinent data transformations(14:67-69). First, it treats **Cut and Splice Strings** as a single process, because the cut and splice data transformations involved always occur together and are interdependent. Second, the DFD includes the Save Best process which returns the solution to the user, thus documenting a previously existing requirement. Finally, it recognizes the requirements for the domain overlay and domain evaluation functions, thereby documenting both a previously existing requirement and a new requirement.

3.2.1 Process Specifications. The process specifications are presented formally as UNITY descriptions in order to avoid implementation and architecture specific details in the requirements analysis(6). The formal process specification for the overall MGA process is shown at Figure 20.

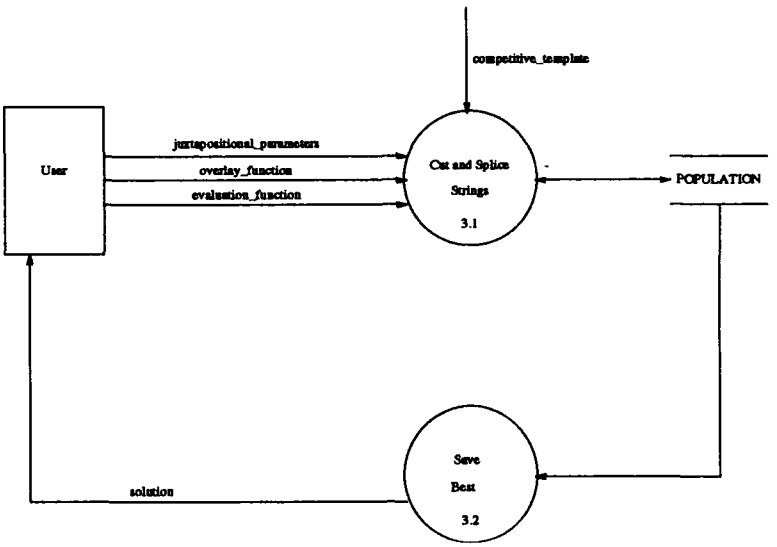

Figure 19. Level 2 Data Flow Diagram for Juxtapositional Phase

3.3 High Level Design.

In order to obtain the maximum benefit from reuse, as much of the original design as possible is retained in the generalized design. Thus, the transaction and transform analysis(43:208-237), which would follow the requirements analysis in the structured design process, are omitted. The existing design is compared to the generalized requirements, and those requirements which are not met are identified. In addition to the capabilities of the original implementation, the generalized implementation must:

- accept a user defined optimization criteria,
- accept a user defined domain initialization function which must specify the cardinality of the encoding scheme,
- internally generate the genic alphabet given the cardinality specified in the domain initialization function, and
- accept a user defined domain overlay function.

These requirements dictate modifications to the messy genetic algorithm's design. Other changes in the requirements which have been discussed previously are clarifications

Program MGA
declare
type *MGA_string* is record {
alleles : array[0..$l_{max} - 1$] of 0..$C - 1$
loci : array[0..$l_{max} - 1$] of 0..$l - 1$
fitness : real
}
type *Population* is list of *MGA_string*
the_best : *MGA_string*
pop : array[0 .. *MAX_GEN*] of *Population*
gen : integer
currpopsize : integer
always
$POP\_SIZE = C^k \binom{l}{k}$
initially
the_best.fitness = *MIN_FITNESS*
gen = 0
conduct_tournament = FALSE
cut_and_splice = FALSE
assign
⟨ *conduct_tournament* := TRUE
if $0 < gen < PRIM\_GEN \wedge \neg$ *conduct_tournament*
⟩ ||
⟨ *cut_and_splice* := TRUE
if $PRIM\_GEN < gen < MAX\_GEN \wedge \neg$ *cut_and_splice* $\wedge$ *conduct_tournament*
⟩ ||
⟨ *conduct_tournament* := TRUE
if $PRIM\_GEN < gen < MAX\_GEN \wedge \neg$ *conduct_tournament* $\wedge$ *cut_and_splice*
⟩ ||
⟨ *the_best* := *pop*[*MAX_GEN* - 1][*i*]
if $gen = MAX\_GEN \wedge \langle \forall j : i < j < popsize[MAX\_GEN - 1] :: pop[MAX\_GEN - 1][i] \rangle$
⟩
end {MGA}

Figure 20. UNITY Description of the Messy Genetic Algorithm

Function *Initialize_Population*
declare
    lock : boolean
    loci : array [0..*k*] of integer
initially
    lock = FALSE
    popindex = 0
    ⟨∀ *i*: $0 \leq i < k$:: *loci*[*i*] = *k* - *i* - 1 ⟩
    ⟨∀ *i*: $0 \leq i <$ *POP_SIZE*:: *distribution*[*i*] = *i* ⟩
    ⟨∀ *i*: $0 \leq i < m$:: *SUB_POP_SIZE*[*i*] =

$$\left\lceil \frac{\binom{l}{k}}{m} + 1 \right\rceil C^k \text{ if } i \leq \left\lceil \binom{l}{k} - 1 \right\rceil \bmod m$$

$$\sim \left\lceil \frac{\binom{l}{k}}{m} \right\rceil C^k \text{ otherwise } \rangle$$

assign
    ⟨[]∀ *i*: $0 \leq i < \binom{l}{k}$::
        ⟨|| ∀ *j*: $0 \leq j < k$ ::
            pop[0][*popindex*].*allele*[*j*],
            pop[0][*popindex*].*loci*[*j*],
            popindex,
            lock :=
            building_block[*i* mod C^k].*allele*[*j*],
            loci[*j*],
            popindex + 1,
            TRUE
        if (*popindex* mod $C^k \neq 0$) ∨ (*lock* = FALSE)
        ⟩ ||
        ⟨|| ∀ *j*: $0 \leq j < k$ ::
            loci[*j*], *lock* :=
            loci[*j*] + 1, FALSE
            if ⟨∀*n*: $1 \leq n < j$ :: *loci*[*n*] = *loci*[*n* - 1] - 1 ⟩
                ∧ *loci*[0] = *l* − 1
                ∧ *loci*[*j*] < *loci*[*j* - 1] - 1
                ∧(*popindex* mod $C^k = 0$)
                ∧(*lock* = TRUE)
        ∼
            0, FALSE
            if ⟨∀*n*: $1 \leq n \leq j$ :: *loci*[*n*] = *loci*[*n* - 1] - 1 ⟩
                ∧(*popindex* mod $C^k = 0$)
                ∧(*lock* = TRUE)
                ∧ *loci*[0] = *l* − 1
        ⟩
    ⟩
end {*Initialize_Population*}

Figure 21. UNITY Description of the Initialization Phase

```
Function GCT
declare
    seq          : integer
    improved     : boolean
    CT           : MGA_string
    variation    : array [0..l − 1, 0..C − 1] of MGA_string
initially
    seq          = 0
    ⟨∀ i: 0 < i < l :: CT.allele[i],CT.loci[i] := Random(0, C − 1), i ⟩
assign
    ⟨ CT.fitness, seq, improved := eval(CT), 1, FALSE ⟩ ⟩
    ||
    ⟨∀ i: 0 < i < l :: ⟨∀ j: 0 < j < C :: variation[i,j] := CT
        if  seq = 0⟩
    ||
    ⟨∀ i: 0 < i < l :: ⟨∀ j: 0 < j < C :: variation[i,j].allele[i], seq := j, 2
        if  seq = 1⟩
    ||
    ⟨[]∀ i: 0 < i < l :: ⟨∀ j: 0 < j < C ::
        CT, improved, seq := variation[i, j], TRUE, 3
        if  better(eval(variation[i,j]), CT.fitness) ∧ seq = 2⟩⟩
    ||
    ⟨ GCT_done, seq := ¬ improved, 0
        if  seq = 3⟩
end {GCT}
```

Figure 22. UNITY Description of the Generate Competitive Template (GCT) Process

```
Function CBB
declare
    seq          : integer
    improved     : boolean
    CT           : MGA_string
    variation    : array [0..l − 1, 0..C − 1] of MGA_string
initially
    seq          = 0
    ⟨∀ i: 0 < i < l ::
        CT.allele[i],CT.loci[i] := Random(0, C − 1), i ⟩
assign
    ⟨∀ i: 0 ≤ i < C^k :: ⟨∀ j: 0 ≤ j < k ::
        building_block[i].allele[j], GBB_done := (i div C^j) mod C, TRUE ⟩⟩
end {CBB}
```

Figure 23. UNITY Description of the Create Building Blocks (CBB) Process

```
Function Conduct_Tournament_Selection
declare
    distribution          : array [1..POP_SIZE] of integer
    permutation           : array [1..POP_SIZE] of integer
    n_sh                  : integer
    θ                     : integer
    popindex              : integer
    candidate_found       : array [1..POP_SIZE] of boolean
    tourn_seq             : integer
always
    cand1 = distribution[popindex]
    cand2 = distribution[(popindex + i mod popindex) + 1]
    θ = ⌈λ1λ2 / l⌉
    compatible = θ ≤ | cand1.loci ∩ cand2.loci |
initially
    tourn_seq = 0
assign
    ⟨|| ∀ i : 1 ≤ i ≤ POP_SIZE ::
        candidate_found[i], permutation[i] n_sh, tourn_seq :=
        FALSE, Random(POP_SIZE) l, 1 ⟩
    ⟩ if tourn_seq = 0 ∧ conduct_tournament
    ||
    ⟨ [] ∀ i : 1 ≤ i ≤ POP_SIZE ::
        distribution[i], distribution[permutation[i]], tourn_seq := distribution[permutation[i]], distribution[i], 2
    ⟩ if tourn_seq = 1
    ||
    ⟨||∀ popindex : 1 ≤ popindex ≤ POP_SIZE ::
        ⟨[]∀ i : 1 ≤ i ≤ n_sh ∧¬candidate_found[popindex] ::
            candidate_found[popindex], newdistribution[popindex], tourn_seq :=
            TRUE, cand1, 3
                if      fitness(cand1) > fitness(cand2) ∨
                        (fitness(cand1) = fitness(cand2) ∧ λ1 < λ2) ∧
                        compatible)
            ~
            TRUE, cand2, 3
                if      fitness(cand2) > fitness(cand1) ∨
                        (fitness(cand1) = fitness(cand2) ∧ λ1 > λ2) ∧
                        compatible)
            ~
            FALSE, cand1, 3
                if      ¬ compatible
        ⟩
    ⟩ if tourn_seq = 2
    ||
    ⟨|| ∀ popindex : 1 ≤ popindex ≤ POP_SIZE ::
        distribution[i], conduct_tournament, tourn_seq := newdistribution[i], FALSE, 0 if candidate_found[i]
    ⟩ if tourn_seq = 3
end {Conduct_Tournament_Selection()}
```

Figure 24. UNITY Description of the Conduct Tournament Process

```
Function Cut_and_Splice
declare
    popindex      : integer
    first_cut     : integer
    second_cut    : integer
    cut_array     : array [1..4] of structure
    cut_index     : integer
    total_strings : integer
    mate1         : structure
    mate2         : structure
    rand3         : float
    c_and_s_seq   : integer
    seq_array     : array [1..POP_SIZE/2] of integer
initially
    cut_index     = 0
    rand3         = rand()
    total_strings = 0
    c_and_s_seq   = 0
    ⟨∀ i : 1 ≤ i ≤ POP_SIZE/2 :: seq_array[i] = 0
always
    mate1 = distribution[popindex]
    mate2 = distribution[popindex + POP_SIZE / 2]
assign
    ⟨|| ∀ i : 1 ≤ i ≤ POP_SIZE ::
        permutation[i] c_and_s_seq :=
        Random(POP_SIZE) 1 ⟩
    ⟩ if c_and_s_seq = 0 ∧ cut_and_splice
    || ⟨ [] ∀ i : 1 ≤ i ≤ POP_SIZE ::
        distribution[i], distribution[permutation[i]], c_and_s_seq := distribution[permutation[i]], distribution[i], 2
    ⟩ if c_and_s_seq = 1
    || ⟨[]∀ popindex : 0 ≤ popindex ≤ POP_SIZE / 2 ::
        ⟨ first_cut, cut_array[0], cut_array[3] :=
            1, head(mate1), tail(mate1) if rand() < p_k(λ_1 − 1)
            ~
            0, mate1, null otherwise
        ⟩ if seq_array[popindex] = 0
        || ⟨ second_cut, cut_array[1], cut_array[2] :=
            1, head(mate2), tail(mate2) if rand() < p_k(λ2 − 1)
            ~
            0, mate2, null otherwise
        ⟩ if seq_array[popindex] = 0
        || ⟨ seq_array[popindex] := 1 if seq_array[popindex] = 0 ⟩
        || ⟨ total_strings, seq_array[popindex] := 2 + first_cut + second_cut, 2 ⟩ if seq_array[popindex] = 1
        || ⟨ population[gen + 1][popindex], popindex, cut_index, c_and_s_seq, seq_array[popindex], cut_and_splice :=
            cat(cut_array[cut_index], cut_array[cut_index + 1]), popindex + 1, cut_index + 2, 0, 0, FALSE
                if cut_index + 1 < total_strings ∧ rand3 > P_s
            ~
            cut_array[cut_index], popindex + 1, cut_index + 1, 0, 0, FALSE
                otherwise
        ⟩ if seq_array[popindex] = 2
    ⟩ if c_and_s_seq = 2
end {Cut_and_Splice}
```

Figure 25. UNITY Description of the Cut and Splice Strings Process

```
Function Save_Best
assign
⟨[]∀ i: 0 ≤ i < POP_SIZE:: the_best := pop[i]
    if pop[i].fitness ¿ the_best.fitness ⟩
end {Save_Best}
```

Figure 26. UNITY Description of the Save Best Process

of existing requirements. They are satisfied by the original implementation, and do not necessitate design changes.

In order to satisfy the optimization criteria requirement, those modules which compare the fitnesses of solutions must be identified, as well as the situations in which they do so. The exact method of satisfying the requirement is left as a low-level design decision. Obvious alternatives are to replace the comparisons with function calls or with macro calls(32:272-274). Function calls are potentially more powerful, while macros are more efficient.

The domain initialization function must be invoked by the MGA code some time after the MGA reads the inputs but before it uses the genic alphabet. Alternatives include invoking the domain initialization function from the input module, the top level executive, the initialization module, or the `Create_Building_Blocks` module. In principle, the function may be implemented either as a macro or as a function call. The determination is left as a low-level design decision.

Likewise, the generation of the genic alphabet must be accomplished after the cardinality is determined, which occurs in the domain initialization function, but before the generation of the building blocks. Alternatives are determined by the choice of when to invoke the domain initialization function, but may include generating the genic alphabet in the domain initialization function itself, the input module, the top level executive, the initialization module, or the `Create_Building_Blocks` module. Again, the determination is left as a low-level design decision.

Finally, the domain overlay function must be invoked prior to every function evaluation. Thus, the modules which invoke the evaluation function must be identified, and

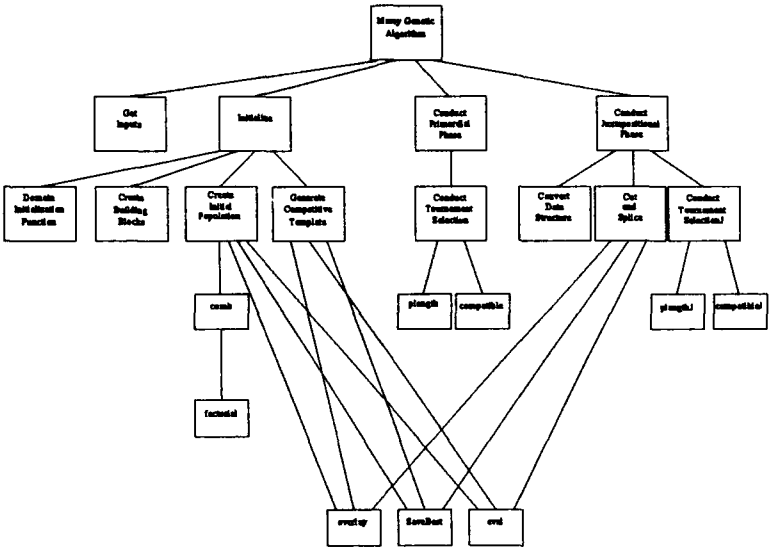

Figure 27. Structure Chart of the Generalized Messy Genetic Algorithm

modified to invoke the overlay function. As is the case with the domain initialization function, the domain overlay function may, in principle, be implemented either as a macro or as a function call. This choice is also left as a low-level design decision.

The structure chart for the generalized MGA is given in Figure 27, while that of the original MGA design(14) is given in Figure 28 for comparison. There are several differences between Figure 28 and the structure chart presented by Dymek(14:106):

- Figure 28 does not show the **pow** C library function,
- Figure 28 documents the fact that the **Generate_Competitive_Template** module invokes the **eval** module,
- Figure 28 includes aesthetic improvements of Dymek's figure.

The generalized MGA structure chart includes the domain initialization and domain overlay modules, which replaces the **process_string** module.

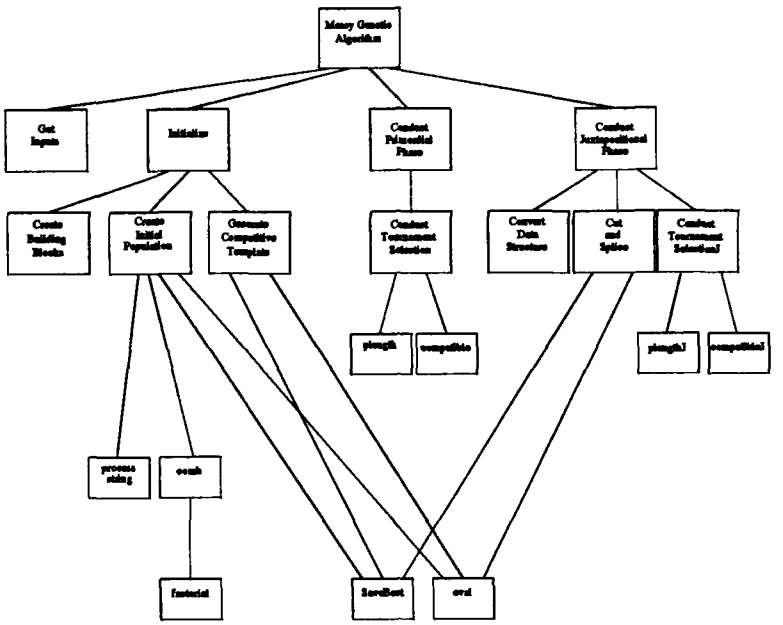

Figure 28. Structure Chart of AFIT's Original Messy Genetic Algorithm

3.4 Low Level Design.

3.4.1 Requirement Driven Low-Level Design Decisions. In order to complete the design of the optimization criteria, the modules which compare solution fitnesses are identified:

- the **Generate_Competitive_Template** function, which compares the candidate templates to select a locally optimal solution, and compares the solutions it generates to the best solution found so far;
- the **Create_Initial_Population** function, which compares each initial solution as it is generated to the best solution found so far in the initial population;
- the **Conduct_Primordial_Phase** function, which compares solutions which are selected to compete against each other; and
- the **Conduct_Juxtapositional_Phase** function, which compares each solution as it is generated to the best solution found so far, and compares solutions which are selected to compete against each other.

Also, the use of a function call or a macro to satisfy the optimization criteria requirement must be addressed in the low-level design. Use of a function call allows greater flexibility than a macro definition. On the other hand, a function call incurs computational overhead not present in a macro definition. Another difference is that if the optimization criterion is defined using a macro definition, modification requires recompilation of all modules which reference the definition, while otherwise only the optimization function must be recompiled. In either case relinking is required. Because it is assumed that most optimization problems of practical interest can be easily posed as either maximization or minimization problems, time efficiency is considered more important than flexibility. Thus, the optimization criterion is defined using a macro. Recompilation and relinking are handled by the UNIX **make** utility(56). All MGA code which compares the fitnesses of two solutions, such as a test **a<b**, is replaced by a macro **better(a,b)**. A macro definition similar to the following, which specifies that fitness **a** is better than fitness **b** if **a<b**, must be included in the domain specific header file **domain.globals.h**:

```
#define better(a,b) a<b
```

The macro definition shown causes the generalized MGA to favor solutions with lower fitnesses, and is therefore appropriate for minimization problems.

It is assumed that the initialization required by many problems is too complex to implement conveniently using a macro definition. Thus, the domain initialization function must be implemented as a C callable function. The selection of the module from which to invoke the domain initialization function is impacted by cohesion considerations(43:82-99). Coupling is not considered, because the existing MGA code makes such extensive use of global variables that coupling is no longer significant. The extensive use of global variables facilitates rapid prototyping and is used commonly in research oriented genetic algorithms(27), but also significantly complicates maintenance(43). Invoking the initialization function from the input function results in procedural cohesion, whereas invoking the function from the executive gives sequential cohesion(43). Sequential cohesion is preferable(43), so that invoking the function from the executive is preferable to invoking it from the input function.

The generalized messy genetic algorithm performs repeated experiments within a single execution. It invokes the initialization and the `Create_Building_Blocks` modules separately for each experiment. Because the domain initialization function need not be invoked separately for each experiment, invoking it from the the initialization or `Create_Building_Blocks` module is inefficient. Thus, the domain initialization function is invoked from the executive.

The modules in which the generation of the genic alphabet may occur are thus limited to the domain initialization function, the executive module, the initialization module, or the `Create_Building_Blocks` module. In order to allow the user the greatest flexibility in the implementation of the evaluation function, the genic alphabet must be generated in the domain initialization function.

The modules which invoke the evaluation function include

- the `Generate_Competitive_Template` function, which evaluates each candidate template;
- the `Create_Initial_Population` function, which evaluates each solution as it is generated; and
- the `Conduct_Juxtapositional_Phase` function, which also evaluates each solution as it is generated.

Each invocation of the evaluation function is preceded by an invocation of the overlay function. In addition, the initial candidate for the competitive template is created by an invocation of the overlay function with an empty template.

3.4.2 Additional Low-Level Design Decisions. A number of additional modifications are included in the generalized version which are not directly traceable to the generalization requirements identified previously, but which facilitate shorter development cycles and/or more efficient experimentation.

1. Code separation. The source code for the original implementation is kept in a single file. In order to facilitate more rapid development and debugging, the generalized

implementation separates the domain independent source code into eight header files and nine source code files. Additionally, the domain specific functions are separated into a header file and three source code files, which reside in a separate directory. This allows several applications to share the domain independent files, thereby simplifying configuration management(29:113-134). The UNIX **make** utility handles all the necessary file processing(56).

2. Modification of the data structure. The original MGA implementation uses two different data structures in the primordial and juxtapositional phase, primarily due to memory considerations(14). The use of different data structures requires the original implementation to use separate functions to overlay the competitive template based upon the current phase. In the generalized version, different data structures in the two phases requires either two overlay functions or a single, more complex function. In order to reduce the burden on the user, the use of different data structures is eliminated. This modification has a number of implications:

   - increased memory requirements,
   - elimination of the need to convert the data structure prior to the juxtapositional phase, and
   - opportunity for the consolidation of large amounts of code.

3. Repeated experiments. Because genetic algorithm experiments typically involve multiple executions with different random number seeds, the generalized implementation accepts as an input parameter the number of experiments to perform. As discussed above, input and initialization are performed one time per execution, prior to the first experiment. Multiple independent experiments are then performed automatically within a single execution.

4. Consolidation of memory management functions. The original MGA implementation performs memory allocation within each module which uses dynamically allocated data structures. The incorporation of repeated experiments in a single execution necessitates modifications to the memory management, because memory need only

be allocated one time per execution. In the generalized implementation, all memory is allocated prior to the beginning of the first experiment. Certain modules must still initialize dynamically allocated memory prior to each experiment.

5. Addition of **measure** function. The GENESIS simple genetic algorithm is designed for use as a research tool(27). As such, it includes a **measure** function, which records solution quality statistics following each generation. The generalized implementation incorporates this function.

3.4.3 Application of the Generalized MGA. Application of the generalized MGA to a particular problem requires the design of the following domain specific algorithms and data structures.

1. The encoding scheme maps between a domain specific representation of the variables being optimized and the MGA representation. Specification of the encoding scheme requires the length of a fully specified solution, the cardinality of the genic alphabet, and the meaning of the gene's alleles for each locus in the string.
2. The domain initialization function defines the cardinality of the problem. It also performs any necessary domain specific initialization.
3. The overlay function. This function takes as input a partial solution and a template, both in MGA representation. The template may be a complete solution or empty. The function must return a fully specified solution. It is generally necessary to first convert the MGA representation of the solution to the domain representation.
4. The evaluation function. This function takes as input a fully specified solution in MGA representation. It must assign a fitness value to the solution. It is generally necessary to first decode the MGA representation to the domain representation.

3.5 Summary.

The requirements analysis and specifications for the generalized messy genetic algorithm, are developed and presented using structured design techniques. The requirements are documented by a problem statement and a hierarchy of balanced data flow diagrams.

The specifications are documented using architecture independent UNITY descriptions. The requirements are compared to the functionality of AFIT's original MGA implementation, and the differences are addressed as design changes. For each new requirement, design alternatives are presented and the design decisions documented. Several additional design features beyond those necessary to meet the requirements are also documented.

IV. Generalized MGA Performance Experiments.

This chapter describes the problems to which the generalized MGA is applied in order to compare its performance to AFIT's original MGA implementation, the simple GA, and the permutation simple GA. It also discusses the manner in which each of the design steps is approached for each of the problems, the design of the performance comparison experiments, and the results of the performance comparisons. The raw experimental data is presented in Appendix A.

In order to assess the capabilities of the generalized MGA, a series of experiments are performed in which the solution quality obtained by the generalized MGA is compared to that obtained by other genetic algorithms. The objectives of the experiments are to examine the generalized MGA's ability to solve

- difficult functional optimization problems,
- NP-complete combinatoric optimization problems,
- problems of practical importance, and
- GA-hard problems.

Deceptive and GA-hard problems, which were discussed previously (Section 2.6), are one of the primary motivations for investigation of the MGA(23, 24, 25).

4.1 Test Problem Selection.

A number of candidate problems are considered, each of which satisfies one or more of the above objectives:

- fully deceptive binary function(14),
- fully deceptive absolute ordering(31),
- fully deceptive relative ordering(31),
- NP-complete problems(18),

- DeJong functions(13),
- polypeptide conformational analysis via minimization of the energy function.

The order-3 fully deceptive binary function problem proposed by Goldberg(20, 23) and modified by Dymek(14) is both a difficult functional optimization problem and GA-hard. Because it has been previously addressed, it allows direct comparison of the results of the generalized MGA to the results of AFIT's original MGA implementation. Also, because it is GA-hard, it clearly demonstrates the improved performance of the MGA relative to the simple GA. This problem is selected as the first test problem (Section 4.2).

The deceptive ordering problems described by Kargupta are order-4 fully deceptive, use string lengths of 32, and have genic alphabets of cardinality 4(31). Both are GA-hard combinatoric optimization problems, and as such are desirable test problems. The initial population for these problems, given by Equation 5, contains $n = 4^4 \begin{pmatrix} 32 \\ 4 \end{pmatrix} = 9.2 \times 10^6$ solutions. Using the minimum overflow factor of 1.0, each solution requires 16 bytes for a record header and the fitness, plus 33 bytes each for alleles and loci, for a total of 82 bytes. Thus, the memory requirement for the initial population for these problems is approximately 720 megabytes. The MGA uses a temporary population in the process of creating each new population, which would require an additional 720 megabytes.

No machine currently available for research at AFIT has this much memory. One alternative to keeping the entire initial population in memory simultaneously are to keep the population on disk. This approach would consume more than the 1226 megabytes available on the largest disk partition. Also, because each member of the population is accessed every generation, such an approach would quickly overwhelm the local area network connecting the Sun workstations to the file servers.

A second alternative is to use a theoretical model of tournament selection to predict the distribution of solutions in the initial juxtapositional phase population. Using this approach, the initialization phase then retains only those solutions which are predicted to receive copies in the juxtapositional phase population. The primordial phase is eliminated. This approach allows the MGA to efficiently solve larger problems, and is expected to

require significantly less execution time due to the elimination of the primordial phase. An accurate model of the tournament selection process is not currently available, so this approach is not currently feasible. The current state of the theoretical model is discussed further in Section 8.2. Because there is no practical way to satisfy or circumvent the memory requirement, the deceptive ordering problems are not investigated in this study.

The Traveling Salesman Problem is a classic NP-complete combinatoric optimization problem. Because any NP-complete problem can be transformed to any other NP-complete problem(18), the generalized MGA's performance on the TSP is representative of its performance for any NP-complete problem. Because the transformation can be performed in polynomial time(18) and the MGA itself is of polynomial time complexity, the MGA can obtain a *semi-optimal* solution to any NP-complete problem in polynomial time. The TSP has been solved previously using a permutation version of the GENESIS simple GA. Use of the TSP as a test problem allows direct comparison of the generalized MGA's performance to that of the permutation GA. It also demonstrates the inability of either the simple GA or AFIT's original MGA to solve permutation problems. The TSP is selected as the second test problem (Section 4.3).

The DeJong functions(13) are non-deceptive functional optimization problems which are frequently used as simple genetic algorithm performance tests. The f2 function, which is also known as Rosenbrock's saddle, is known to be difficult for simple GAs, although it is not GA-hard(15). It is also of practical importance, because it is representative of a class of functions found in aircraft and missile control systems(14). Use of the f2 function as a test problem allows direct comparison of the generalized MGA's performance to that of AFIT's original MGA and the simple GA on a difficult functional optimization problem. This function is selected as the third test problem (Section 4.4).

The conformational analysis problem is representative of a class of computationally difficult problems, and is also of interest due to its importance in the design of materials with specific non-linear optical properties. The essence of the problem is the minimization of a molecule's energy(39:296), which is usually modeled empirically by the nonlinear

function

$$
\begin{aligned}
V(r_1, r_2, ..., r_N) \;=\; & \sum_{bonds} \frac{1}{2} K_b (R - R_0)^2 + \\
& \sum_{angles} \frac{1}{2} K_a (\theta - \theta_0)^2 + \\
& \sum_{dihedrals} K_d [1 + \cos(n\phi - \gamma)] + \\
& \sum_{i,j} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \frac{q_i q_j}{\epsilon r_{ij}}
\end{aligned} \tag{6}
$$

Significant research(16, 42, 48, 59) has been invested in efficient and accurate evaluation of Equation 6. In order to benefit from this research, it is desirable to use the energy function from an existing program as the evaluation function for the MGA. In order to do so, the function must be callable from a C program. No such function is currently available at AFIT (but see Chapter VIII for future research recommendations).

Thus, three problems are selected as test problems for these experiments. The fully deceptive binary function optimization problem tests the performance of the MGA when solving a GA-hard functional optimization problem (Section 4.2). The TSP tests the MGA's performance when applied to an NP-complete combinatoric optimization problem (Section 4.3). Rosenbrock's Saddle is used to test the performance of the generalized MGA when it is used to solve a difficult functional optimization problem (Section 4.4).

For each selected problem, the parameters used for each GA are specified, and the methodology used in selecting the parameters is described. Each of the problems is solved using the generalized MGA, Dymek's original MGA implementation(14), and Grefenstette's GENESIS simple GA(27). In addition, the TSP is solved using the permutation version of GENESIS(27). The results obtained using each of the GAs are compared on the basis of solution quality, which is defined appropriately for each of the problems. All of the genetic algorithms are implemented on a Sun-4 in C (Sun Release 4.1) under the SunOS operating system (Release 4.1.2). The C language was chosen for the generalized MGA in order to allow reuse of the code from AFIT's original implementation.

In order to minimize the effects of the random number generator seed, the number of executions is selected to be large enough to assure that the results are statistically significant. Following Dymek(14), each experiment is arbitrarily performed 40 times for each GA. In each case, all 40 generalized MGA experiments are performed first using a single seed. The generalized MGA performs repeated experiments in a single execution, and records the seeds used by each of the experiments. These seeds are then used for the corresponding experiments of the original MGA. This ensures that the two MGAs generate the same competitive template and conduct identical tournaments during the primordial phase. Side by side execution under the `dbxtool` debugger verifies that this occurs. Slightly different behavior is observed during the juxtapositional phase beginning in the cut and splice operation of the second generation[1].

The simple GA uses the random number sequence for different purposes than the MGA does. For example, the MGA uses the first ℓ random numbers to initialize the competitive template, while the simple GA uses them to initialize the first member of the initial population. Thus, it would be meaningless for the simple GA to use the same random number sequence as the MGA.

4.2 Deceptive Binary Problem.

The first test problem is the order-3 fully deceptive binary functional optimization problem addressed by Dymek(14). The problem consists of ten 3-bit subproblems. Each subproblem is order-3 fully deceptive, and is described by the mapping in Table 4. The total fitness of a solution to the full problem is the sum of the fitnesses of the solutions to the subproblems.

The encoding scheme for the function is based on a string of thirty genes and a binary genic alphabet, as defined by Goldberg(23). The bits corresponding to a particular subproblem are separated within the string, as shown in Table 5. Separation of the genes increases the defining length of important building blocks, thus making the deceptive problem GA-hard.

[1]This is currently unexplained, but may be due to the correction of a bug in the original implementation.

Alleles	Fitness
000	28
001	26
010	22
011	0
100	14
101	0
110	0
111	30

Table 4. Order 3 Deceptive Function Subproblem Fitnesses

Subproblem	Loci		
1	1	6	11
2	2	7	12
3	3	8	13
4	4	9	14
5	5	10	15
6	16	21	26
7	17	22	27
8	18	23	28
9	19	24	29
10	20	25	30

Table 5. Order 3 Deceptive Function Subproblems

Parameter	Value
Random Seed	123456789
Experiments	40
String Length	30
Block Size (1 - String_Length)	3
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	3
Total Reductions	4
Shuffle Number (>1)	30
Cut Probability	0.0166667
Splice Probability	1.0
Total_Generations	19
Overflow (>1.0)	1.6
Selection method (T/P)	T

Table 6. Fully Deceptive Function Generalized MGA Input Parameters

The domain initialization function defines the cardinality of the genic alphabet to be 2, corresponding to a binary genic alphabet. The overlay function accepts as input a partial solution and a competitive template, both in MGA representation. It produces a fully specified solution by selecting genes corresponding to unspecified loci first from the partial solution, then from the competitive template.

The evaluation function accepts as input a fully specified solution in MGA representation. It returns the fitness of the solution, which is simply the sum of the fitness contributions from each subproblem.

4.2.1 Experimental Design. The parameters for the generalized MGA experiment are shown in Table 6. The encoding scheme described earlier determines the string length and genic alphabet cardinality. The MGA ignores the genic alphabet parameter. The block size is selected to match the known level of deception in the function.

The reduction rate, reduction interval, and total number of reductions are selected to correspond as closely as possible to those used by Goldberg(23:513-514). Goldberg reports that for this problem the population size, which begins at 32,480, "is cut in half every other generation ... until it reaches size $n = 2030$ [and] the primordial phase is terminated ... after generation 11." This indicates a total of four reductions, either in generations 1,

3, 5, and 7 or 2, 4, 6, and 8. In either case, additional primordial phase generations are specified after the last reduction. Both AFIT's original implementation and the generalized implementation terminate the primordial phase after the last reduction. To ensure that the total number of reductions and the total number of primordial phase tournaments are as close as possible to those used by Goldberg, a reduction interval of 3 is selected.

Following Goldberg, a shuffle number equal to string length is used(24:427). Similarly, the bitwise cut probability is chosen as a function $p_k = \frac{1}{2\ell}$ of the string length, which results in an overall cut probability of $p_c = p_k \ell = 0.5$ for a fully specified string. Goldberg states that this "corresponds roughly" to the crossover probability of 1.0 used for the simple genetic algorithm(23:514). This seeming discrepancy is most likely due to a desire to foster rapid string growth in the juxtapositional phase. Deb shows that when $p_k \ll 1$, the average string length almost doubles every generation(10). Therefore, in the initial generations of this experiment, when $p_c \ll 1$ for most strings in the population, the average string length grows rapidly. In the fourth and later generations, the average string length is such that $p_c \approx 1.0$ for most strings in the population, and string growth is minimal. Similarly, the splice probability for this experiment is chosen to match that used by Goldberg, which is in turn based on the crossover probability used for the simple genetic algorithm.

Goldberg used a total of 30 generations, of which 11 were in the primordial phase. Therefore "Total_Generations" parameter, which specifies the number of *juxtapositional* generations, is chosen to be 19. The overflow factor specifies the maximum ratio by which actual string length may exceed nominal string length. Goldberg estimates the probability of a previously expressed schema not being expressed given placement in the back of a spliced pair as

$$P(N|B) \leq 1 - \left(\frac{1-k}{\ell}\right)^{\lambda^*} \tag{7}$$

where N is the event that the schema is not expressed, B is the event that the schema is placed at the back of a spliced pair, k is the block size, λ^* is the maximum string length in the population, and ℓ is the nominal string length. Figure 29 shows that for the block size and string length associated with this problem, overflow factors $\lambda^*/\ell > 1.6$, result in less

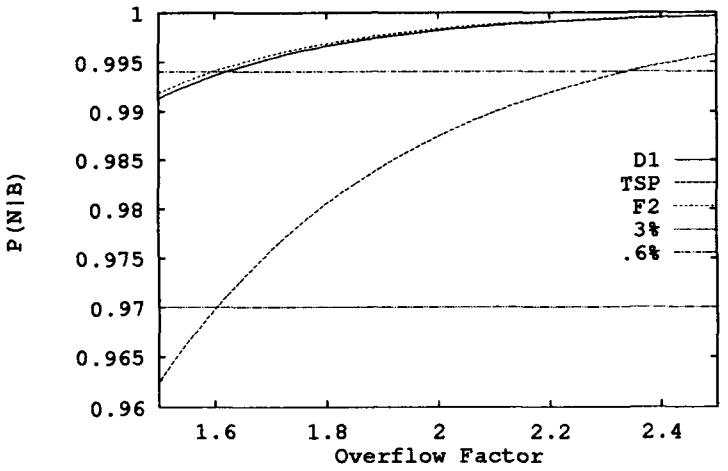

Figure 29. Probability of Non-expression of Schema as a Function of Overflow Factor ($k/\ell = 0.1$)

than a 1% chance that a schema is not expressed when it is placed at the end of a spliced pair. Dymek uses an overflow factor of 1.6, which is shown to provide a low probability of schema loss. Larger overflow factors do not significantly reduce the probability, and smaller overflow factors do not significantly reduce either the memory requirement or the execution time. Because there is no compelling reason to choose a different value, Dymek's value is adopted for this experiment.

With the exception of the reduction interval and the genic alphabet, the parameters for the original MGA experiments are the same as those for the generalized MGA. In the original MGA the reduction interval specifies the number of non-reducing tournaments per reduction *excluding* the reducing tournament, while in the generalized MGA it specifies the number of tournaments per reduction *including* the reducing tournament. Thus, to achieve the same behavior, the reduction interval for the original MGA must be specified to be one less than that for the generalized MGA.

Unlike the generalized MGA, the original MGA uses the genic alphabet parameter. The cardinality is assigned the length of the string used to define the genic alphabet. For this experiment, the cardinality is 2. The characters used to define the genic alphabet are

Parameter	Value
String Length	30
Block Size (1 - String_Length)	3
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	2
Total Reductions	4
Shuffle Number (>1)	30
Cut Probability	0.0166667
Splice Probability	1.0
Total_Generations	19
Overflow (>1.0)	1.6
Selection method (T/P)	T

Table 7. Fully Deceptive Function Original MGA Input Parameters

arbitrary. This experiment uses the characters "0" and "1," which are intuitively appealing because of their common use in boolean logics.

The relevant parameters for the simple genetic algorithm experiment, which are shown at Figure 8, are chosen to allow as direct a comparison as possible to the messy genetic algorithm. The population size is chosen to match the juxtapositional population size of the MGA. The total number of trials is

$$T = N_i + N_f G \tag{8}$$

where N_i is the initial population size, N_f is the juxtapositional population size, and G is the number of juxtapositional generations. The primordial phase contribution is due only to the initial population because later primordial generations do not evaluate new solutions.

Following Goldberg, the crossover rate is chosen to be 1.0, and the mutation rate is chosen to be 0.0. Because we are concerned only with the overall best solution quality obtained as opposed to the average solution quality over time, a generation gap of 1.0 is chosen. This is consistent with other "generational" genetic algorithm experiments.

GENESIS is designed to minimize functions, and modification to perform maximization requires changes to multiple modules. Such modification unnecessarily introduces

Parameter	Value
Experiments	40
Total Trials	71050
Population Size	2030
Structure Length	30
Crossover Rate	1.000
Mutation Rate	0.000
Generation Gap	1.000
Random Seed	123456789

Table 8. Fully Deceptive Function Simple GA Input Parameters

Alleles	Fitness
000	2
001	4
010	8
011	30
100	16
101	30
110	30
111	0

Table 9. Order 3 Deceptive Function Subproblem Fitnesses

significant risk of programmer error. Rather than modify GENESIS to perform function maximization, a modified version of the test function is minimized. The simplicity of the test function and greater familiarity with the code imply a lower risk of programmer error. Subproblems of the modified function have the fitnesses shown in Table 9, which are equal to the original fitnesses subtracted from the maximum fitness. Thus, the modified function has a global minimum of 0, in contrast to the original function's maximum of 300. The final solutions for each experiment are subtracted from 300 to allow direct comparison with solutions obtained using other methods.

4.2.2 Results. The mean solution found by the generalized messy genetic algorithm has a fitness of 297.15, while that of the original messy genetic algorithm has a fitness of 298.20. The Kruskal-Wallis H test for these samples has an H-value of 5.671875, which indicates statistical significance at the 2.5% significance level. Therefore, the generalized

messy genetic algorithm does not perform as well as the original implementation for this problem. After adjusting the solutions obtained by GENESIS to allow comparison with the messy genetic algorithm data, the mean solution has a fitness of 285.35, and the Kruskal-Wallis H test comparing them to the generalized MGA solutions yields an H-value of 57.567801, which is sufficient to indicate statistical significance at the 0.5% level. The generalized messy genetic algorithm clearly outperforms the simple genetic algorithm on this problem.

4.3 Combinatoric Optimization: Traveling Salesman Problem.

An instance of the TSP consists of a complete graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where each $v_i \in \mathcal{V} = \{1, 2, 3, ..., N\}$ is a node representing a city, and each $c_{ij} \in \mathcal{E}$ is an edge having a corresponding weight d_{ij} representing the distance between two cities. A solution to the TSP is a permutation $\mathcal{P} = p_1 p_2 p_3 ... p_N$ of the vertices, representing a *tour* such that the total cost of the tour

$$C = \sum_{i=1}^{N} d_{i,p_i} \tag{9}$$

is minimized, subject to the additional constraint that the subgraph $\mathcal{G}'$ containing the edges $\mathcal{E}' = \{e'_{ip_i} : 1 \leq i \leq N\}$ consists of a single cycle.

The encoding scheme for the TSP is based upon a string of length N and a genic alphabet of cardinality N. Each gene in the string represents an element p_i in a tour, where i is the gene's locus. The gene's allele j identifies the city which follows city i in the tour. The domain initialization function loads or randomly generates a matrix containing the distances d_{ij}. It also assigns the cardinality of the genic alphabet to be equal to N.

The overlay function takes as input a partial tour and a template, both in MGA representation, and produces a fully specified and valid solution. In order to be valid, the specified tour must include each city exactly once, and consist of exactly one cycle. An example of a tour which satisfies the first constraint but not the second is shown in Figure 30. The algorithm used is an extension to the nearest neighbor algorithm taken from Grefenstette's GENESIS TSP program which ensures that a tour contains each city exactly once(27). The high level statement of the algorithm is presented in Figure 31.

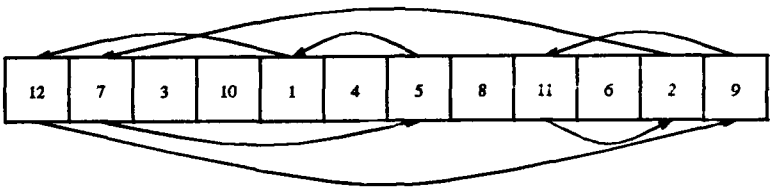

Figure 30. Invalid TSP Tour Containing Multiple Cycles

1. **freecount** $\leftarrow N-1$;
2. $\forall i : 1 \leq i \leq N ::$ **freelist**[i] $\leftarrow i$, **where**[i] $\leftarrow i$, **tour**[i] $\leftarrow -1$
3. Include partial tour in **tour**
4. if (**freecount** $= 0$) exit;
5. Include competitive template in **tour**
6. if (**freecount** > 0) use **freelist** to complete valid tour

Figure 31. TSP Overlay Function Algorithm

In this algorithm, the fully specified tour is built in **tour**, **freecount** is the number of unvisited cities currently in **tour**, **freelist** is a list of the unvisited cities, and **where** is an index into **freelist**. The details of step 3 are shown in Figure 32. Essentially, the algorithm examines each gene in the partial tour to determine whether or not it can be added to the tour without violating one of the above constraints. In order to do so, it must ensure that the destination city specified by the gene is not already included in the tour, and that adding the edge from the source city to the destination city does not result in a cycle of length less than N. Each time a city is added to **tour**, it is removed from **freelist**, which is then compressed. Step 5 of the high level algorithm is very similar to step 3, with the obvious difference that all references to the partial tour are replaced by references to the competitive template. The details of step 6 are shown in Figure 33. The algorithm selects an arbitrary city from **freelist** to be the "start" city, and removes it from **freelist**. It then follows the path defined by **tour** beginning at the start city until it finds an unspecified city. It again selects an arbitrary city from **freelist** to be the "destination" city, adds it to **tour**, and removes it from **freelist**. This is repeated until no more cities remain in **freelist**, at which time the "start" city is added to **tour** in the last remaining position.

For each **gene** $\in$ **partial_tour**,

1. **source** $\leftarrow$ **gene.locus**, **dest** $\leftarrow$ **gene.allele**
2. If (**dest** $\neq$ **source**) $\wedge\neg$(**dest** $\in$ **tour**)
   (a) **city** $\leftarrow$ **tour[dest]**, $k \leftarrow 0$
   (b) Repeat **city** $\leftarrow$ **tour[city]**, $k \leftarrow k+1$ until $\neg$(**tour[city]** $\in \{1..N\}$)$\vee$ (**city** = **source**)
   (c) If $\neg$(**tour[city]** $\in \{1..N\}$) $\vee$ ($k = N-1$) remove **dest** from **freelist** (and compress **freelist**)
      i. **tour[source]** $\leftarrow$ **dest**
      ii. **index** $\leftarrow$ **where[dest]**
      iii. **lastcity** $\leftarrow$ **freelist[freecount]**
      iv. **freecount** $\leftarrow$ **freecount** -1
      v. **freelist[index]** $\leftarrow$ **lastcity**
      vi. **where[lastcity]** $\leftarrow$ **index**

Figure 32. Include Partial Tour Algorithm

The evaluation function for the TSP takes as input a fully specified solution in MGA representation and returns the fitness of the solution, as given by Equation 9.

4.3.1 Experimental Design. The specific instance of the TSP selected for the performance experiment has 12 cities, which is the largest problem for which sufficient memory is available to contain the resulting initial population. The distance matrix is neither symmetric nor Euclidean, so the validity of the experimental results is not limited to problems in those classes. The parameters for the generalized MGA experiment are shown at Table 10. Again, the encoding scheme determines the string length and cardinality. The domain initialization function for the TSP application of the generalized MGA assigns a cardinality of 12, corresponding to the number of cities. The block size is chosen to be the suspected order of deception in the problem. It is very unlikely that an arbitrary instance of the TSP is fully deceptive at order 3 or higher. On the other hand, a block size of 1 results in a very small initial population size, given by Equation 5 as $n = 12^1 \left(\begin{array}{c} 12 \\ 1 \end{array} \right) = 144$. Therefore, a block size of 2 is chosen.

1. **start** ← **freelist**[0];
2. Remove **start** from **freelist** (and compress **freelist**)
   (a) **index** ← **where**[**start**]
   (b) **lastcity** ← **freelist**[**freecount**]
   (c) **freecount** ← **freecount** −1
   (d) **freelist**[**index**] ← **lastcity**
   (e) **where**[**lastcity**] ← **index**
   (f) **where**[**start**] ← −1
3. **dest** ← **start**, **source** ← **dest**
4. Repeat **dest** ← **tour**[**source**], **source** ← **dest** until ¬(**tour**[**source**] ∈ {1..N})
5. while (**freecount** > 0)
   (a) $l \leftarrow 0$
   (b) Repeat **dest** ← **freelist**[l], $l \leftarrow l + 1$ until **dest** ≠ **source**
   (c) Remove **dest** from **freelist** (and compress **freelist**)
      i. **index** ← **where**[**dest**]
      ii. **lastcity** ← **freelist**[**freecount**]
      iii. **freecount** ← **freecount** −1
      iv. **freelist**[**index**] ← **lastcity**
      v. **where**[**lastcity**] ← **index**
      vi. **where**[**dest**] ← −1
   (d) **tour**[**source**] ← **dest**;
   (e) Repeat **dest** ← **tour**[**source**], **source** ← **dest** until ¬(**tour**[**source**] ∈ {1..N})
6. **tour**[**dest**] ← **start**;

Figure 33. Complete Tour Algorithm

Parameter	Value
Random Seed	123456789
Experiments	40
String Length	12
Block Size (1 - String_Length)	2
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	3
Total Reductions	2
Shuffle Number (>1)	12
Cut Probability	0.0416667
Splice Probability	1.0
Total_Generations	24
Overflow (>1.0)	2.4
Selection method (T/P)	T

Table 10. TSP Generalized MGA Input Parameters

In this application, the initial population size $n = 12^2 \begin{pmatrix} 12 \\ 2 \end{pmatrix} = 9504$ is smaller than that of the fully deceptive binary function. Only 2 reductions are required to obtain a juxtapositional population size of the same order as that used in the previous experiment.

Again, a shuffle number equal to string length is used. The cut probability is again chosen to be $p_k = \frac{1}{2\ell}$, and the splice probability is held at 1.0. An overflow factor of 2.4, obtained from Figure 29, is used to maintain the same probability of schema expression used in previous experiments.

Goldberg does not discuss the selection of the reduction interval. In order to select an appropriate selection interval, a preliminary experiment is performed, in which the reduction interval is varied from 1 to 4 and the "Total_Generations" parameter is adjusted to hold the total of primordial and juxtapositional generations fixed at 30. The remaining parameters are held constant at the values specified above. Initial experiments in which the generalized MGA is executed 40 times for each reduction interval failed to result in statistically significant differences in solution quality. In order to increase the probability of detecting any such differences, the experiment is repeated, and the generalized MGA executed 400 times for each interval. The mean tour lengths resulting from each of the reduction intervals are given at Table 11, along with the optimal tour length.

Reduction Interval	Mean Tour Length
1	1474.8075
2	1488.9300
3	1478.6100
4	1507.6275
Optimal	1410.0000

Table 11. Preliminary TSP Experiment Mean Tour Lengths

Reduction Intervals	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1, 2	2	3.8415	2.422517	No
1, 3	2	3.8415	0.013097	No
1, 4	2	3.8415	9.020059	Yes
2, 3	2	3.8415	2.551348	No
2, 4	2	3.8415	2.249045	No
3, 4	2	3.8415	9.020059	Yes
1, 2, 3	3	5.9915	3.324822	No
1, 2, 4	3	5.9915	9.130928	Yes
1, 3, 4	3	5.9915	12.037575	Yes
2, 3, 4	3	5.9915	9.215521	Yes
All	4	7.8147	12.642195	Yes

Table 12. Preliminary TSP Experiment Kruskal-Wallis H Test Results

In order to determine whether or not the observed differences in solution quality can be considered significant, the results are compared using the Kruskal-Wallis H Test. A total of eleven tests are performed in order to identify one reduction interval which results in statistically better performance than the others. Each independent sample is compared to the other three in turn, the samples are tested in groups of three, and all four samples are tested together. The experimental data are compared at the 5% level of significance, which gives a critical values of $\chi^2_{1,0.05} = 3.8415$ for pairwise comparisons, $\chi^2_{2,0.05} = 5.9915$ for comparisons involving three of the independent samples, and $\chi^2_{3,0.05} = 7.8147$ for the comparison involving all four of the samples. The results of the comparisons are given at Table 12. The test rejects the null hypothesis only in cases involving a reduction interval of 4, which results in statistically *worse* solutions. Therefore, we can safely choose the reduction interval to be either 1, 2, or 3. To maintain consisitency with the deceptive binary experiments, a reduction interval of 3 is used.

Parameter	Value
Random Seed	123456789
Experiments	40
String Length	12
Block Size (1 - String_Length)	2
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	2
Total Reductions	3
Shuffle Number (>1)	12
Cut Probability	0.0416667
Splice Probability	1.0
Total_Generations	24
Overflow (>1.0)	2.4
Selection method (T/P)	T

Table 13. TSP Original MGA Input Parameters

The parameters for the experiment using AFIT's original MGA implementation are again based on those used in the generalized implementation (See Table 13). In order to produce the same number of primordial generations per reduction, the reduction interval for the original implementation is specified to be 2. The remaining parameters are identical.

The absence of an overlay function in the original MGA implementation allows evaluated strings to represent invalid solutions to the TSP problem. A common approach to optimizing constrained problems with genetic algorithms is the use of penalty functions(21:85-86). In this experiment, a penalty is incorporated into the fitness function which assigns a very poor fitness to any string which does not represent a valid solution. No distinction is made based upon how "close" a string is to representing a valid solution. Thus, every invalid solution has the same fitness, which is worse than the fitness of any valid solution.

The parameters for the simple genetic algorithm experiments, given in Table 14, are also chosen to produce behavior as close as possible to the messy genetic algorithm's. The population size is chosen to match the messy GA's juxtapositional population size, and the number of trials is determined from Equation 8 to be $9504 + 2376 \times 24 = 66528$. The remaining parameters are unchanged from the deceptive binary function experiment. The parameters for the experiments using the permutation version of the simple genetic algorithm are identical to those for the experiments using the standard version. Both

Parameter	Value
Experiments	40
Total Trials	66528
Population Size	2376
Structure Length	12
Crossover Rate	1.000
Mutation Rate	0.000
Generation Gap	1.000
Scaling Window	5
Report Interval	2000
Structures Saved	1
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	lc
Random Seed	123456789

Table 14. TSP Simple GA Input Parameters

simple genetic algorithm experiments use the same fitness function, including the penalty for invalid solutions, which the original MGA experiment uses.

4.3.2 Results. The best solutions found by the generalized messy genetic algorithm have a mean fitness of 1460.300, where the optimal tour length is 1410. None of the experiments using either the original MGA or the simple GA identify valid solutions. The generalized MGA clearly outperforms both the original MGA and the simple GA on this problem.

The best solutions found by the permutation GA have a mean fitness of 2046.275. The Kruskal-Wallis H test comparing the solution quality of the generalized MGA to that of the permutation GA has an H-value of 59.259259, which indicates statistical signficance at the 0.5% level. The generalized messy genetic algorithm clearly outperforms the permutation genetic algorithm on this problem. This may be due to the relatively small problem size, and should not interpreted as an indication that the same would be true for large instances of the TSP or other permutation problems.

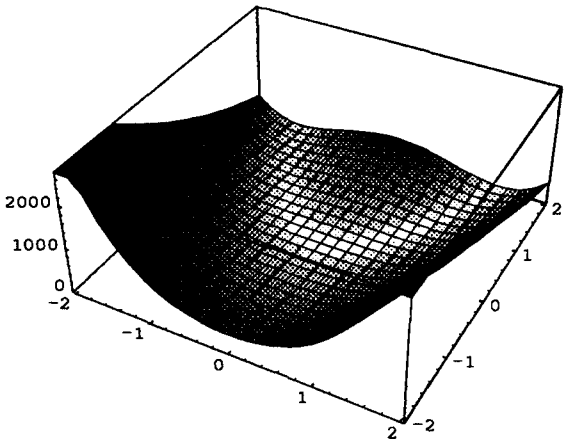

Figure 34. Linear Plot of Rosenbrock's Saddle

4.4 Functional Optimization: Rosenbrock's Saddle.

Rosenbrock's Saddle, also known as DeJong function f2, is one of five classical GA test functions identified by DeJong(13). The function is a non-linear, non-convex function of 2 variables, having a deep parabolic valley. It is described by the equation

$$f2 = 100(x^2 - y)^2 - (1 - x)^2 \tag{10}$$

A plot of the function is given at Figure 34. The first term of the function is symmetric about the x-axis, having a minimum at points satisfying the relation $y = x^2$. The second term, the contribution of which is much smaller than the first term, has a minimum of zero only at the point $x = 1$. The function has one minimum at the point $x = 1$, $y = 1$. The differences in the values along the parabola are not very large, especially relative to the maximum values (poor solutions).

4.4.1 Experimental Design. Following Dymek, the encoding scheme for the Rosenbrock's Saddle application is based upon a string of length 24 and a binary genic alphabet (14:33). Each of the independent variables is restricted to the range between -2.048 and

Parameter	Value
Random Seed	123456789
Experiments	40
String Length	24
Block Size (1 - String_Length)	3
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	1
Total Reductions	3
Shuffle Number (>1)	24
Cut Probability	0.0208333
Splice Probability	1.0
Total_Generations	27
Overflow (>1.0)	1.6
Selection method (T/P)	T

Table 15. Rosenbrock's Saddle Generalized MGA Input Parameters

2.047 and discretized in even increments of 0.001. Each is then encoded as a 12-bit binary string.

The domain initialization function defines the cardinality of the genic alphabet to be 2, corresponding to a binary genic alphabet. The overlay function accepts as input a partial solution and a competitive template, both in MGA representation. It produces a fully specified solution by selecting genes corresponding to unspecified loci first from the partial solution, then from the competitive template.

The evaluation function takes as input a fully specified solution in MGA representation. It returns the fitness of the solution, as determined by Equation 9.

As with the deceptive binary problem and the combinatoric optimization problem, the encoding scheme for the functional optimization problem determines the string length and cardinality for the functional optimization experiments using the generalized messy genetic algorithm. The remaining parameters, shown at Table 19, are selected in similar fashion to the previous experiments. The domain initialization function, independent of the genic alphabet specified in the input file, generates a binary genic alphabet and assigns a cardinality of 2. Rosenbrock's saddle is known to not be deceptive(15), so the block size is selected based only on initial population size considerations. Population size as a function of block size for this encoding scheme is shown at Table 16. In order to produce

Block Size	Population Size
1	48
2	1,104
3	16,192
4	170,016

Table 16. Initial Population Size as a Function of Block Size

Reduction Interval	Mean Solution Fitness
1	0.0087898675
2	0.0974485850
3	0.0395790650
4	0.1151387764

Table 17. Preliminary Functional Optimization Experiment Mean Solution Fitnesses

an initial population size of the same magnitude as that used in previous experiments, a block size of 3 is selected. Three reductions are then required to obtain a juxtapositional population size in the typical range. The shuffle number and the cut probability are once again chosen based on string length, and the splice probability is held at 1.0. An overflow factor of 1.6 is sufficient to provide negligible probability of schema loss (See Figure 29).

A preliminary experiment similar to the one performed for the TSP experiment is performed to select a reduction interval. The reduction interval is again varied from 1 to 4, the "Total_Generations" parameter adjusted to maintain a total of 30 primordial and juxtapositional generations, and the remaining parameters held constant at the values specified above. In order to ensure statistically significant results, each experiment is conducted 400 times. The mean solution fitnesses are given at Table 17. As discussed above, the optimal solution to Rosenbrock's Saddle has a fitness of 0.

The results of the preliminary experiment are again compared using the Kruskal-Wallis H Test at the 5% level of significance. The results of the comparisons are given at Table 18. The test rejects the null hypothesis in every case, indicating that different reduction intervals result in statistically different solution populations. Because a reduction interval of 1 results in the lowest mean, this value is used in further experiments.

Reduction Intervals	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1, 2	2	3.8415	418.450955	Yes
1, 3	2	3.8415	263.864261	Yes
1, 4	2	3.8415	490.984549	Yes
2, 3	2	3.8415	163.141650	Yes
2, 4	2	3.8415	163.141650	Yes
3, 4	2	3.8415	267.795592	Yes
1, 2, 3	3	5.9915	547.580796	Yes
1, 2, 4	3	5.9915	615.372328	Yes
1, 3, 4	3	5.9915	657.768561	Yes
2, 3, 4	3	5.9915	297.245356	Yes
All	4	7.8147	786.600884	Yes

Table 18. Preliminary Rosenbrock's Saddle Experiment Kruskal-Wallis H Test Results

The parameters for the original MGA experiment are once again based on those used in the generalized MGA experiment (See Table 19). The reduction interval for the original implementation is specified to be 0, which specifies that every primordial generation includes a population reduction, as in the generalized experiment. The remaining parameters are identical.

The parameters for the simple genetic algorithm experiments are given in Table 20. The population size matches the messy GA's juxtapositional population size, and the number of trials is calculated using Equation 8 to be $16192 + 2024 \times 27 = 70840$. The remaining parameters are unchanged from the deceptive binary function and TSP experiments.

4.4.2 Results. The best solutions found by the generalized messy genetic algorithm have a mean fitness of 0.0083415, where the optimal is 0. The best solutions found by the original messy genetic algorithm have a mean fitness of 0.2737781951. The Kruskal-Wallis H test for these samples has an H-value of 18.595382, which indicates statistical significance at the 0.5% significance level. Therefore, the generalized messy genetic algorithm outperforms the original implementation for this problem.

The best solutions found by the simple genetic algorithm have a mean fitness of 0.0003242341, and the Kruskal-Wallis H test comparing them to the generalized MGA solutions yields an H-value of 48.267037, which is sufficient to indicate statistical significance

Parameter	Value
Random Seed	123456789
Experiments	40
String Length	24
Block Size (1 - String_Length)	3
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	0
Total Reductions	3
Shuffle Number (>1)	24
Cut Probability	0.0208333
Splice Probability	1.0
Total_Generations	27
Overflow (>1.0)	1.6
Selection method (T/P)	T

Table 19. Rosenbrock's Saddle Original MGA Input Parameters

Parameter	Value
Experiments	40
Total Trials	70840
Population Size	2024
Structure Length	24
Crossover Rate	1.000
Mutation Rate	0.000
Generation Gap	1.000
Scaling Window	5
Report Interval	2000
Structures Saved	1
Max Gens w/o Eval	2
Dump Interval	0
Dumps Saved	0
Options	cel
Random Seed	123456789

Table 20. Rosenbrock's Saddle Simple GA Input Parameters

Experiment	Original MGA	Simple GA	Permutation GA
Deceptive Binary Problem	Marginally Outperforms Generalized MGA	Generalized MGA Outperforms	N/A
Combinatoric Optimization Problem	Unable to Find Valid Solution	Unable to Find Valid Solution	Generalized MGA Outperforms
Functional Optimization Problem	Generalized MGA Outperforms	Outperforms Generalized MGA	N/A

Table 21. Summary of Relative Performance

at the 0.5% level. The simple genetic algorithm clearly outperforms the generalized messy genetic algorithm on this problem. Two possible causes of the MGA's poorer performance are

- the presence of nonuniform building block sizes(24), and
- the high degree of convergence obtained via tournament selection in the primordial phase.

Further investigation of these possibilities is recommended.

4.5 Summary.

Experiments are performed which compare the performance of the generalized MGA to that of AFIT's original MGA and to the GENESIS simple GA. The results of those experiments are summarized in Table 21. The generalized MGA obtains better solutions than the other three GAs with two exceptions. First, the original MGA performs marginally better on the deceptive binary function problem. Second, the simple GA performs better on DeJong function f2. The former is possibly a side effect of a correction of a bug in the generalized implementation cut and splice operator which affects the frequency with which the cut operator is applied. The latter is possibly due to non-uniform building block size and/or scaling. Both cases merit further investigation.

V. Parallelization of the Messy Genetic Algorithm.

Dymek's measurements(14) show that for some problems, as much as 87% of execution time of the messy genetic algorithm is spent in the primordial phase, indicating that reasonable speedup demands parallelization of the primordial phase. This chapter describes four parallel implementations of the messy genetic algorithm, each of which exploits the data parallelism present in the primordial phase (Section 5.1). It also describes experiments comparing the execution time and solution quality of the four implementations (Section 5.2), and the results of those experiments (Section 5.3).

5.1 Initial Population Distribution Strategies.

As discussed in Section 2.7, parallel implementation of an algorithm can be accomplished by mapping its UNITY description to the particular architecture in question(6). Because the parallelization effort in this study focuses on the primordial phase, the mapping begins with an examination of the *Conduct_Tournament_Selection* process specification (See Figure 24). The assign section of the process description contains four groups of statements. The first group describes the initialization of the *candidate_found* and *permutation* data structures. The second describes the use of the *permutation* data structure to "shuffle" the *distribution* data structure. The third group of statements describes the tournament selection itself, in which the *newdistribution* data structure is built from the *distribution* and *permutation* data structures. The final group of statements copies the *newdistribution* data structure back to the *distribution* data structure to prepare for the next tournament.

An arbitrary statement S_i, $1 \leq i \leq POP\_SIZE$, from the second group, assigned to processor P_j, $1 \leq j \leq m$, may reference any variable in the *distribution* data structure, depending upon the random initialization of the *permutation* data structure. Variables must be allocated to the same processor as any statements which reference them. Therefore, all of the variables in the *distribution* data structure must be allocated to P_j. The remaining statements from the second group may also reference variables from the *distribution* data structure, and consequently must be allocated to P_j. By similar reasoning, all of the vari-

ables in the *permutation* data structure must be allocated to P_j. Each of the statements belonging to one of the other three groups references at least one variable belonging to either the *distribution* or the *permutation* data structure. Therefore, they must also be allocated to P_j. Likewise, the remaining variables referenced by those statements must be allocated to P_j.

While the tournament selection algorithm is shown to be inherently sequential, approximations exist which are parallelizable. The modified algorithm is obtained by first arbitrarily allocating the variables of the original algorithm to processors and then modifying the statements in such a way that each statement references only variables which are allocated to a single processor. Thus, tournaments are conducted only between solutions for which the *distribution* variables are allocated to the same processor. Because there are many ways to arbitrarily allocate the variables to the processors, there are many possible parallel tournament selection algorithms.

By observing that the mapping between the *distribution* data structure and the actual solutions is determined by the *Initialize_Population* process, it can be shown that all parallel tourament selection algorithms which are designed in this manner are functionally equivalent. The UNITY description for one such algorithm is shown at Figure 35. This algorithm may be directly mapped to a distributed memory architecture by allocating to the same processor those statements which share the same value of j.

The number of *distribution* variables, and therefore solutions, allocated to each processor is determined by the *SUB_POP_SIZE* data structure, which is initialized by the *Initialize_Population* process. The actual solutions which are assigned to the same subpopulation is determined by the mapping from the *distribution* data structure to the *pop* data structure, which is also defined by the *Initialize_Population* process. The remainder of this section presents four versions of the *Initialize_Population* process, each of which describes a different partitioning and mapping of solutions to the processors.

The first strategy, called the "indexed" strategy, is discussed by Dymek but not implemented. It assigns building blocks to processors using an interleaving scheme, using the block's first defined locus as the interleaving key. The UNITY description for this

```
Function Conduct_Parallel_Tournament_Selection
declare
    distribution          : array [1..SUB_POP_SIZE[j]] of integer
    permutation           : array [1..SUB_POP_SIZE[j]] of integer
    n_sh                  : integer
    θ                     : integer
    popindex              : integer
    candidate_found       : array [1..SUB_POP_SIZE[j]] of boolean
    tourn_seq             : integer
always
    cand1 = distribution[SUB_POP_SIZE[j] + popindex]
    cand2 = distribution[SUB_POP_SIZE[j] + ((popindex + i mod SUB_POP_SIZE[j]) + 1)]
    θ = ⌈λ1λ2 / l⌉
    compatible = θ ≤ | cand1.loci ∩ cand2.loci |
    SUB_POP_START[j] = Σ_{i=0}^{j-1} SUB_POP_SIZE[j]
initially
    tourn_seq = 0
assign

    ⟨|| ∀ j . 1 ≤ j ≤ P ::
        ⟨|| ∀ i : 1 ≤ i ≤ SUB_POP_SIZE[j] ::
            candidate_found[SUB_POP_START[j] + i], permutation[SUB_POP_START[j] + i] n_sh, tourn_seq :=
            FALSE, Random(POP_SIZE) l, 1 ⟩
        ⟩ if tourn_seq = 0 ∧ conduct_tournament
        ||
        ⟨ [] ∀ i : 1 ≤ i ≤ SUB_POP_SIZE[j] ::
            distribution[SUB_POP_START[j] + i], distribution[permutation[SUB_POP_START[j] + i]], tourn_seq :=
            distribution[permutation[SUB_POP_START[j] + i]], distribution[SUB_POP_START[j] + i], 2
        ⟩ if tourn_seq = 1
        ||
        ⟨||∀ popindex : 1 ≤ popindex ≤ SUB_POP_SIZE[j] ::
            ⟨[]∀ i : 1 ≤ i ≤ n_sh ∧¬candidate_found[popindex] ::
                candidate_found[popindex], newdistribution[popindex], tourn_seq :=
                TRUE, cand1, 3
                    if      (fitness(cand1) > fitness(cand2) ∨
                             (fitness(cand1) = fitness(cand2))∧ λ1 < λ2) ∧
                             compatible)
                ~
                TRUE, cand2, 3
                    if      (fitness(cand2) > fitness(cand1) ∨
                             (fitness(cand1) = fitness(cand2))∧ λ1 > λ2) ∧
                             compatible)
                ~
                FALSE, cand1, 3
                    if      ¬ compatible
            ⟩
        ⟩ if tourn_seq = 2
        ||
        ⟨|| ∀ i : 1 ≤ i ≤ SUB_POP_SIZE[j] ::
            distribution[SUB_POP_START[j] + i], conduct_tournament, tourn_seq :=
            newdistribution[SUB_POP_START[j] + i], FALSE, 0 if candidate_found[SUB_POP_START[j] + i]
        ⟩ if tourn_seq = 3
    ⟩
end {Conduct_Parallel_Tournament_Selection()}
```

Figure 35. UNITY Description of the Parallel Conduct Tournament Process

Nodes	0	1	2	3	4	5	6	7	Variation
1	32480	-	-	-	-	-	-	-	-
2	19584	12896	-	-	-	-	-	-	52%
4	10752	8832	7168	5728	-	-	-	-	88%
8	5632	5120	4640	4192	3776	3392	3032	2696	109%

Table 22. Indexed Distribution Strategy Allocation

strategy is presented at Figure 36. Using this strategy on m processors, each processor j, $0 \leq j < m$, is allocated

$$N_j = \sum_{i=0}^{I} \begin{pmatrix} l-j-im-1 \\ k-1 \end{pmatrix} \tag{11}$$

solutions, where l is the string length, k is the block size, and

$$I = \lceil (l-j)/m \rceil - 1. \tag{12}$$

Thus, use of this strategy for a 30-bit problem with a block size of 3, allocates solutions to processors as shown in Table 22. This strategy allocates significantly more solutions to some processors than to others. In cases where the number of processors is greater than the string length, some processors are not allocated any solutions. Based upon these observations, Dymek rejects this strategy because of anticipated poor load balancing(14).

The second strategy, "modified indexed" distribution, is also discussed by Dymek but not implemented. It is a modification of the first strategy in which the first defined locus is greater than the number of processors are assigned to processors in reverse order. The UNITY description for this strategy is presented at Figure 37. Using this strategy on m processors, each processor j, $0 \leq j < m$, is allocated

$$N_j = \begin{pmatrix} l-j-1 \\ k-1 \end{pmatrix} + \sum_{i=1}^{I} \begin{pmatrix} l-(i+1)m+j-1 \\ k-1 \end{pmatrix} \tag{13}$$

solutions. Thus, use of this strategy for the same problem allocates solutions to processors as shown in Table 23. This strategy allocates solutions more evenly than the indexed strategy, although there is still some imbalance. In cases where the number of processors

```
Function Initialize_Population
declare
    lock        : boolean
    loci        : array [0..k] of integer
initially
    lock        = FALSE
    popindex    = 0
    ⟨∀ i: 0 ≤ i < k:: loci[i] = k - i - 1 ⟩
    ⟨∀ i: 0 ≤ i < POP_SIZE:: distribution[i] = i ⟩
always
    I           = ⌈(l − j)/m⌉ − 1
    ⟨∀ j: 0 ≤ j < m:: SUB_POP_SIZE[j] =
                  ∑_{i=0}^{I} ( l − j − im − 1 choose k − 1 )
assign
    ⟨[]∀ i: 0 ≤ i < ( l choose k ) ::
        ⟨
            ⟨|| ∀ j: 0 ≤ j < k ::
                pop[0][popindex].allele[j],
                pop[0][popindex].loci[j] :=
                building_block[i mod C^k].allele[j],
                loci[j]
            ⟩
            || ⟨ distribution[subpopindex[loci[0]]], subpopindex[loci[0]], lock, :=
            subpopindex[loci[0]], subpopindex[loci[0]] + 1, TRUE ⟩
        ⟩ if (popindex mod C^k ≠ 0) ∨ (lock = FALSE)
        || ⟨|| ∀ j: 0 ≤ j < k ::
            loci[j], lock :=
            loci[j] + 1, FALSE
            if   ⟨∀n: 1 ≤ n < j :: loci[n] = loci[n - 1] - 1 ⟩
                 ∧ loci[0] = l − 1
                 ∧ loci[j] < loci[j - 1] - 1
                 ∧(popindex mod C^k = 0)
                 ∧(lock = TRUE)
        ~
            0, FALSE
            if   ⟨∀n: 1 ≤ n ≤ j :: loci[n] = loci[n - 1] - 1 ⟩
                 ∧(popindex mod C^k = 0)
                 ∧(lock = TRUE)
                 ∧ loci[0] = l − 1
        ⟩
    ⟩
end {Initialize_Population}
```

Figure 36. Indexed Initialize Population Process

Nodes	0	1	2	3	4	5	6	7	Variation
1	32480	-	-	-	-	-	-	-	-
2	15624	16856	-	-	-	-	-	-	8%
4	7536	7888	8296	8760	-	-	-	-	16%
8	4096	4032	3992	3976	3992	4040	4120	4232	6%

Table 23. Modified Indexed Distribution Strategy Allocation

Function *Initialize_Population*
declare
    lock : boolean
    loci : array [0..k] of integer
initially
    lock = FALSE
    popindex = 0
    $\langle \forall\, i\colon 0 \le i < k :: loci[i] = k - i - 1 \rangle$
    $\langle \forall\, i\colon 0 \le i < POP\_SIZE :: distribution[i] = i \rangle$
always
    I $= \lceil (l-j)/m \rceil - 1$
    $\langle \forall\, j\colon 0 \le j < m :: SUB\_POP\_SIZE[j] = \binom{l-j-1}{k-1} + \sum_{i=1}^{I} \binom{l-(i+1)m+j-1}{k-1} \rangle$
assign
    $\langle [] \forall\, i\colon 0 \le i < \binom{l}{k} ::$
        $\langle$
            $\langle \| \forall\, j\colon 0 \le j < k ::$
                pop[0][*popindex*].*allele*[*j*],
                pop[0][*popindex*].*loci*[*j*] :=
                building_block[i mod C^k].*allele*[*j*],
                loci[*j*]
            $\rangle$
            $\| \langle$ *distribution*[*subpopindex*[*loci*[0]]], *subpopindex*[*loci*[0]], *lock*, :=
            subpopindex[*loci*[0]], *subpopindex*[*loci*[0]] + 1, TRUE $\rangle$
        $\rangle$ if (*popindex* mod $C^k \neq 0$) $\vee$ (*lock* = FALSE)
        $\| \langle \langle \| \forall\, j\colon 0 \le j < k ::$
            loci[*j*], *lock* :=
            loci[*j*] + 1, FALSE
            if $\langle \forall n\colon 1 \le n < j :: loci[n] = loci[n-1] - 1 \rangle$
                $\wedge$ *loci*[0] = $l - 1$
                $\wedge$ *loci*[*j*] < *loci*[*j* - 1] - 1
                $\wedge$(*popindex* mod $C^k = 0$)
                $\wedge$(*lock* = TRUE)
        $\sim$
            0, FALSE
            if $\langle \forall n\colon 1 \le n \le j :: loci[n] = loci[n-1] - 1 \rangle$
                $\wedge$(*popindex* mod $C^k = 0$)
                $\wedge$(*lock* = TRUE)
                $\wedge$ *loci*[0] = $l - 1$
        $\rangle$
    $\rangle$
end {*Initialize_Population*}

Figure 37. Modified Indexed Initialize Population Process

Nodes	0	1	2	3	4	5	6	7	Variation
1	32480	-	-	-	-	-	-	-	-
2	16240	16240	-	-	-	-	-	-	0%
4	8120	8120	8120	8120	-	-	-	-	0%
8	4060	4060	4060	4060	4060	4060	4060	4060	0%

Table 24. Interleaved/Block Distribution Strategy Allocations

is greater than half of the string length, this strategy approaches the indexed strategy. Dymek also rejects this strategy because of anticipated poor load balancing(14).

The third strategy imposes an ordering on the building blocks, then interleaves the building blocks across the processors based upon their position in the ordering. It is called the "interleaved" distribution strategy. The UNITY description for this strategy is presented at Figure 38. Dymek selected this strategy because it achieves "almost perfect" load balancing based strictly upon the number of building blocks assigned to each processor. Using this strategy on m processors, each processor j, $0 \leq j < m$, is allocated

$$N_j = \left\lceil \frac{\binom{l}{k} - j}{m} \right\rceil \tag{14}$$

solutions. Thus, this strategy allocates solutions to processors for the example problem as shown in Table 24. This strategy allocates solutions as evenly as possible. Processors receive differing numbers of solutions only in cases where the number solutions in the initial population is not evenly divisible by the number of processors. In such cases, subpopulation sizes differ by 1 solution. This is the strategy which is used in Dymek's parallel MGA implementation.

The last strategy, which Dymek does not discuss, assigns the building blocks to the processors using a block distribution strategy. It is called the "block" distribution strategy. The UNITY description for this strategy is presented at Figure 39. This strategy allocates the same number of solutions to each processor as the interleaved strategy.

Function *Initialize_Population*
declare
    lock : boolean
    loci : array [0..*k*] of integer
initially
    lock = FALSE
    popindex = 0
    ⟨∀ *i*: $0 \leq i < k$:: *loci*[*i*] = *k* - *i* - 1 ⟩
    ⟨∀ *i*: $0 \leq i <$ *POP_SIZE*:: *distribution*[*i*] = *i* ⟩
always
    I = $\lceil (l - j)/m \rceil - 1$
    ⟨∀ *j*: $0 \leq j < m$:: *SUB_POP_SIZE*[*j*] =
    $$\left\lceil \frac{\binom{l}{k} - j}{m} \right\rceil$$
assign
    ⟨[]∀ *i*: $0 \leq i < \binom{l}{k}$ ::
        ⟨
            ⟨|| ∀ *j*: $0 \leq j < k$ ::
                pop[0][*popindex*].*allele*[*j*],
                pop[0][*popindex*].*loci*[*j*] :=
                building_block[*i* mod C^k].*allele*[*j*],
                loci[*j*]
            ⟩
            || ⟨ *distribution*[*subpopindex*[*i* mod C^k]], *subpopindex*[*i* mod C^k], *lock*, :=
            subpopindex[*i* mod C^k], *subpopindex*[*i* mod C^k] + 1, TRUE ⟩
        ⟩ if (*popindex* mod $C^k \neq 0$) ∨ (*lock* = FALSE)
        || ⟨|| ∀ *j*: $0 \leq j < k$ ::
            loci[*j*], *lock* :=
            loci[*j*] + 1, FALSE
            if ⟨∀*n*: $1 \leq n < j$ :: *loci*[*n*] = *loci*[*n* - 1] - 1 ⟩
                ∧ *loci*[0] = *l* − 1
                ∧ *loci*[*j*] < *loci*[*j* - 1] - 1
                ∧(*popindex* mod $C^k = 0$)
                ∧(*lock* = TRUE)
        ~
            0, FALSE
            if ⟨∀*n*: $1 \leq n \leq j$ :: *loci*[*n*] = *loci*[*n* - 1] - 1 ⟩
                ∧(*popindex* mod $C^k = 0$)
                ∧(*lock* = TRUE)
                ∧ *loci*[0] = *l* − 1
        ⟩
    ⟩
end {*Initialize_Population*}

Figure 38. Interleaved Initialize Population Process

```
Function Initialize_Population
declare
    lock        : boolean
    loci        : array [0..k] of integer
initially
    lock        = FALSE
    popindex    = 0
    ⟨∀ i: 0 ≤ i < k:: loci[i] = k - i - 1 ⟩
    ⟨∀ i: 0 ≤ i < POP_SIZE:: distribution[i] = i ⟩
always
    I           = ⌈(l − j)/m⌉ − 1
    ⟨∀ j: 0 ≤ j < m:: SUB_POP_SIZE[j] =
                  ⌈ ( l choose k ) − j / m ⌉
assign
    ⟨[]∀ i: 0 ≤ i < ( l choose k ) ::
        ⟨
            ⟨|| ∀ j: 0 ≤ j < k ::
                pop[0][popindex].allele[j],
                pop[0][popindex].loci[j] :=
                building_block[i mod C^k].allele[j],
                loci[j]
            ⟩
            || ⟨ distribution[subpopindex [⌊im/POP_SIZE⌋], subpopindex[⌊im/POP_SIZE⌋], lock, :=
            subpopindex[⌊im/POP_SIZE⌋], subpopindex[⌊im/POP_SIZE⌋] + 1, TRUE ⟩
        ⟩ if (popindex mod C^k ≠ 0) ∨ (lock = FALSE)
        || ⟨|| ∀ j: 0 ≤ j < k ::
            loci[j], lock :=
            loci[j] + 1, FALSE
            if  ⟨∀n: 1 ≤ n < j :: loci[n] = loci[n - 1] - 1 ⟩
                ∧ loci[0] = l − 1
                ∧ loci[j] < loci[j - 1] - 1
                ∧(popindex mod C^k = 0)
                ∧(lock = TRUE)
        ~
            0, FALSE
            if  ⟨∀n: 1 ≤ n ≤ j :: loci[n] = loci[n - 1] - 1 ⟩
                ∧(popindex mod C^k = 0)
                ∧(lock = TRUE)
                ∧ loci[0] = l − 1
        ⟩
    ⟩
end {Initialize_Population}
```

Figure 39. Block Initialize Population Process

An illustration contrasting the distributions resulting from the application of the four strategies to a problem with a string length of 10 on a 4-processor architecture is shown at Tables 25 and 26.

Dymek's selection of the interleaved distribution strategy is apparently based upon an unstated assumption that the execution time of a given processor in the primordial phase is a function only of the number of solutions allocated to the processor. This assumption ignores the details involved in the selection of the second mate for each tournament. As shown in Figures 18 and 35, the selection of the second mate involves a loop, in which strings are randomly selected until either one is found which is "compatible" with the first mate or the shuffle size has been exceeded. As discussed in Section 3.1, the shuffle size is an input parameter specified at run time. Thus, the execution time of the primordial phase on a given processor is a function of

- the number of solutions allocated to the processor's subpopulation,
- the probability with which two solutions randomly selected from a particular subpopulation are compatible, and
- the shuffle size.

The data distribution strategy affects both the number of solutions allocated to each subpopulation and the probability of compatibility. The effect on compatibility is complex, but Tables 25 and 26 give some indication of the four strategies behaviors with respect to compatibility. It is apparent that the indexed, modified indexed, and block strategies tend to allocate compatible solutions to the same processor, while the interleaved distribution does not.

5.2 Experimental Design.

In order to determine the effects of each of the data distribution strategies on solution quality and execution time, a series of experiments are performed. Versions of the parallel messy genetic algorithm(14) which use modified data distribution strategies are implemented on an 8-node iPSC/2 in C (Green Hill's C-386 compiler Version 1.8.4) under the UNIX System V/386 Release 3.2 operating system.

Node				
Indexed	Modified Indexed	Block	Interleaved	Defining Loci
0	0	0	0	1 2 3
			1	1 2 4
			2	1 2 5
			3	1 2 6
			0	1 2 7
			1	1 2 8
			2	1 2 9
			3	1 2 10
			0	1 3 4
			1	1 3 5
			2	1 3 6
			3	1 3 7
			0	1 3 8
			1	1 3 9
			2	1 3 10
			3	1 4 5
			0	1 4 6
			1	1 4 7
			2	1 4 8
			3	1 4 9
			0	1 4 10
			1	1 5 6
			2	1 5 7
			3	1 5 8
			0	1 5 9
			1	1 5 10
			2	1 6 7
			3	1 6 8
			0	1 6 9
			1	1 6 10
		1	2	1 7 8
			3	1 7 9
			0	1 7 10
			1	1 8 9
			2	1 8 10
			3	1 9 10
1	1	1	0	2 3 4
			1	2 3 5
			2	2 3 6
			3	2 3 7
			0	2 3 8
			1	2 3 9
			2	2 3 10
			3	2 4 5
			0	2 4 6
			1	2 4 7
			2	2 4 8
			3	2 4 9
			0	2 4 10
			1	2 5 6
			2	2 5 7
			3	2 5 8
			0	2 5 9
			1	2 5 10
			2	2 6 7
			3	2 6 8
			0	2 6 9
			1	2 6 10
			2	2 7 8
			3	2 8 9
		2	0	2 7 10
			1	2 8 9
			2	2 8 10
			3	2 9 10

Table 25. Distribution Strategies

Node				
Indexed	Modified Indexed	Block	Interleaved	Defining Loci
2	2	2	0	3 4 5
			1	3 4 6
			2	3 4 7
			3	3 4 8
			0	3 4 9
			1	3 4 10
			2	3 5 6
			3	3 5 7
			0	3 5 8
			1	3 5 9
			2	3 5 10
			3	3 6 7
			0	3 6 8
			1	3 6 9
			2	3 6 10
			3	3 7 8
			0	3 7 9
			1	3 7 10
			2	3 8 9
			3	3 8 10
			0	3 9 10
3	3	2	1	4 5 6
			2	4 5 7
			3	4 5 8
			0	4 5 9
			1	4 5 10
		3	2	4 6 7
			3	4 6 8
			0	4 6 9
			1	4 6 10
			2	4 7 8
			3	4 7 9
			0	4 7 10
			1	4 8 9
			2	4 8 10
			3	4 9 10
0	3	3	0	5 6 7
			1	5 6 8
			2	5 6 9
			3	5 6 10
			0	5 7 8
			1	5 7 9
			2	5 7 10
			3	5 8 9
			0	5 8 10
			1	5 9 10
1	2	3	2	6 7 8
			3	6 7 9
			0	6 7 10
			1	6 8 9
			2	6 8 10
			3	6 9 10
2	1	3	0	7 8 9
			1	7 8 10
			2	7 9 10
3	0	3	3	8 9 10

Table 26. Distribution Strategies (cont.)

Parameter	Value
String Length	30
Block Size (1 - String_Length)	3
Genic Alphabet	01
Reduction Rate (0 - 1.0)	0.5
Reduction Interval	2
Total Reductions	4
Shuffle Number (>1)	30
Cut Probability	0.0166667
Splice Probability	1.0
Total_Generations	19
Overflow (>1.0)	1.6
Selection method (T/P)	T

Table 27. Fully Deceptive Function Parallel MGA Input Parameters

For each strategy, the problem solved is the fully deceptive binary function optimization problem described in Section 4.2. The deceptive function is selected as the target problem in order to determine whether or not any of the distribution strategies affect the parallel MGA's ability to solve such problems. Additional test problems, such as the combinatoric and functional optimization problems discussed in Sections 4.3 and 4.4 are not addressed, although such experiments are recommended.

The 8-node iPSC/2 allocates processors in sets of 1, 2, 4, or 8. Each MGA implementation is executed 10 times for each of the four possible hypercube dimensions, using the first 10 random number seeds generated in the experiments described in Section 4.2. The number of experiments is selected to be large enough to give a reasonable chance of obtaining results which are statistically significant at the 1% level.

The parameters used in the executions are shown at Table 27. They are the same parameters used for the original MGA implementation experiments described in Section 4.2.

5.3 Results.

The average solution quality obtained using each of the four distribution strategies for each hypercube dimension is shown at Table 28. Kruskal-Wallis Tests at the 5% level of significance for each hypercube dimension are shown at Table 29. The tests indicate that

Distribution Strategy	Processors	Solution Quality	
		Mean	Standard Deviation
Indexed	1	297.8	1.751
	2	298.2	1.135
	4	297.6	1.265
	8	296.6	1.647
Modified Indexed	1	297.8	1.751
	2	297.0	2.160
	4	298.0	1.333
	8	296.8	1.932
Interleaved	1	297.2	2.150
	2	298.0	1.633
	4	298.2	1.989
	8	297.8	1.751
Block	1	297.2	2.150
	2	298.0	1.886
	4	298.2	1.751
	8	297.8	1.476

Table 28. Parallel MGA Solution Quality Results

Processors	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1	4	7.8147	0.749268	No
2	4	7.8147	1.935000	No
4	4	7.8147	1.233659	No
8	4	7.8147	3.411585	No

Table 29. Parallel MGA Solution Quality Kruskal-Wallis Tests

choice of distribution strategy does not have a statistically significant effect on solution quality for the fully deceptive binary function using small hypercube dimensions.

The execution times and speedups for each significant operation are given in Appendix B for each version of the parallel MGA. As expected, the execution times for the generation of the competitive template and the creation of the building blocks are very similar for all of the distribution strategies and all hypercube dimensions. Kruskal Wallis Tests confirm that neither choice of distribution strategy nor hypercube dimension significantly affects execution time spent in these operations. The H values of the tests are 8.612818 and 5.291925 respectively, which are both less than $\chi^2_{15,0.05} = 24.996$.

Distribution Strategies	k	$\chi^2_{k-1,0.005}$	H value	Reject H_0?
1, 2	2	7.8794	14.285714	Yes
1, 3	2	7.8794	14.285714	Yes
1, 4	2	7.8794	14.285714	Yes
2, 3	2	7.8794	14.285714	Yes
2, 4	2	7.8794	14.285714	Yes
3, 4	2	7.8794	12.091429	Yes

Table 30. Initialization Time Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Distribution Strategies	k	$\chi^2_{k-1,0.005}$	H value	Reject H_0?
1, 2	2	7.8794	14.285714	Yes
1, 3	2	7.8794	14.285714	Yes
1, 4	2	7.8794	14.285714	Yes
2, 3	2	7.8794	14.285714	Yes
2, 4	2	7.8794	14.285714	Yes
3, 4	2	7.8794	14.285714	Yes

Table 31. Primordial Phase Execution Time Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Both distribution strategy and hypercube dimension significantly affect the execution times of the population initialization, the primordial phase, and the data structure conversion operations. The results of Kruskal-Wallis Tests comparing the data for execution times using 8 processors are summarized in Tables 30 through 34. The results indicate statistical evidence at the 0.5% level of significance that the choice of distribution strategy affects the execution time spent in these operations. Tests at the 5% level of significance do not indicate that distribution strategy affects juxtapositional phase or overall execution time.

Hypercube dimension also significantly affects the execution times of the population initialization, the primordial phase, and the data structure conversion operations. Average speedups are shown for each implementation for the primordial phase alone (Figure 40), the three distribution strategy dependent operations together (Figure 41), and the overall execution (Figure 42). Kruskal-Wallis Tests are performed comparing speedups for the

Distribution Strategies	k	$\chi^2_{k-1,0.005}$	H value	Reject H_0?
1, 2	2	7.8794	14.285714	Yes
1, 3	2	7.8794	14.285714	Yes
1, 4	2	7.8794	14.285714	Yes
2, 3	2	7.8794	14.285714	Yes
2, 4	2	7.8794	14.285714	Yes
3, 4	2	7.8794	14.285714	Yes

Table 32. Data Structure Conversion Time Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Distribution Strategies	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1, 2	2	3.8415	0.365714	No
1, 3	2	3.8415	2.520000	No
1, 4	2	3.8415	0.142857	No
2, 3	2	3.8415	3.291429	No
2, 4	2	3.8415	1.651429	No
3, 4	2	3.8415	0.462857	No
All	4	7.8147	4.251220	No

Table 33. Juxtapositional Phase Execution Time Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Distribution Strategies	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1, 2	2	3.8415	0.022857	No
1, 3	2	3.8415	0.012857	No
1, 4	2	3.8415	1.285714	No
2, 3	2	3.8415	0.005714	No
2, 4	2	3.8415	0.822857	No
3, 4	2	3.8415	0.822857	No
All	4	7.8147	1.519390	No

Table 34. Total Execution Time Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

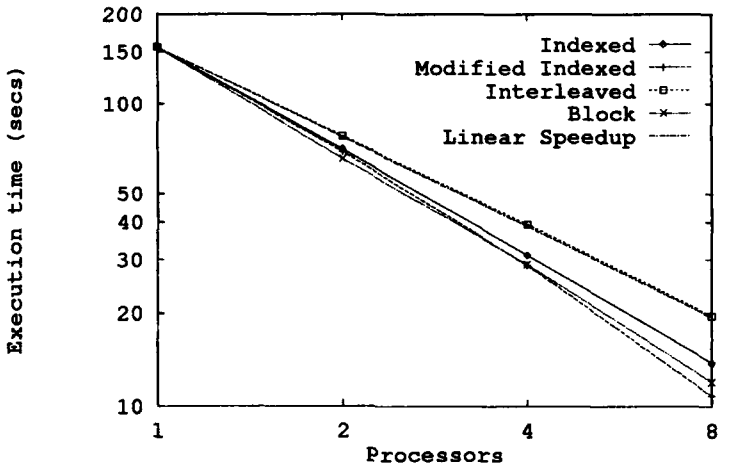

Figure 40. Primordial Phase Speedup

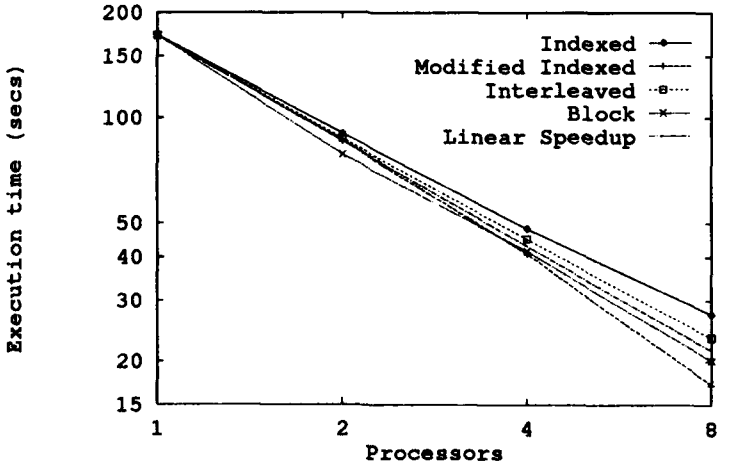

Figure 41. Distribution Strategy Dependent Operations Speedup

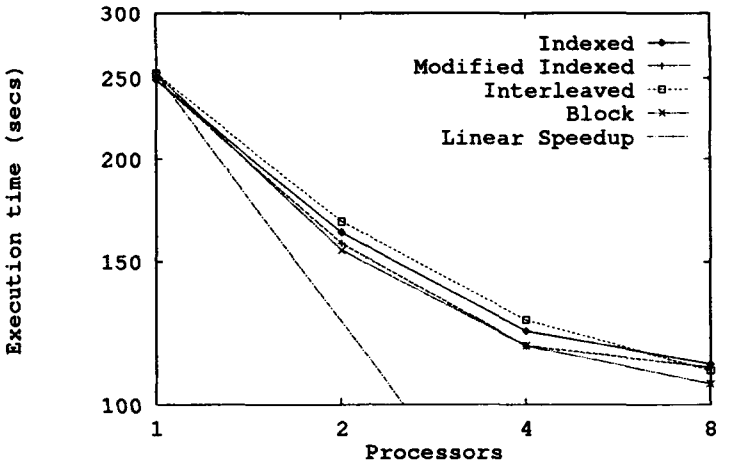

Figure 42. Overall Speedup

Distribution Strategies	k	$\chi^2_{k-1,0.005}$	H value	Reject H_0?
1, 2	2	7.8794	14.285714	Yes
1, 3	2	7.8794	14.285714	Yes
1, 4	2	7.8794	14.285714	Yes
2, 3	2	7.8794	14.285714	Yes
2, 4	2	7.8794	14.285714	Yes
3, 4	2	7.8794	14.285714	Yes

Table 35. Primordial Phase Speedup Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

primordial phase alone (Table 35), the three distribution strategy dependent operations together (Table 35), and the overall execution (Table 35) using 8 processors. The results indicate statistical evidence at the 0.5% level of significance that the choice of distribution strategy affects the speedup in the the execution of the primordial phase, and the conversion of the data structures for the juxtapositional phase. Thus, the modified indexed strategy yields the best speedup of the primordial phase and the parallelized portion. The block strategy results in the next best speedup. As discussed previously, the modified indexed strategy is expected to exhibit significantly worse speedup for implementations for which the number of processors exceeds one half of the string length. Thus, the block strategy is expected to give the best primordial phase speedup in such implementations.

Distribution Strategies	k	$\chi^2_{k-1,0.005}$	H value	Reject H_0?
1, 2	2	7.8794	14.285714	Yes
1, 3	2	7.8794	14.285714	Yes
1, 4	2	7.8794	14.285714	Yes
2, 3	2	7.8794	14.285714	Yes
2, 4	2	7.8794	14.285714	Yes
3, 4	2	7.8794	14.285714	Yes

Table 36. Parallelized Operations Speedup Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Distribution Strategies	k	$\chi^2_{k-1,0.05}$	H value	Reject H_0?
1, 2	2	3.8415	0.005714	No
1, 3	2	3.8415	0.280000	No
1, 4	2	3.8415	2.520000	No
2, 3	2	3.8415	0.142857	No
2, 4	2	3.8415	2.285714	No
3, 4	2	3.8415	0.965714	No
All	4	7.8147	3.162439	No

Table 37. Overall Speedup Kruskal-Wallis Tests

Distribution Strategy Legend: 1 = Indexed, 2 = Modified Indexed, 3 = Interleaved, 4 = Block

Tests at the 5% level of significance do not indicate that the distribution strategy affects juxtapositional phase or overall execution speedup. The shape of the overall speedup curves does suggest that the block strategy may provide significantly better speedup for implementations using more processors.

The variations in initialization phase execution times are attributable to load imbalance. Both the indexed and modified indexed strategies require some processors to initialize more solutions that other processors. The small difference between the interleaving strategy initialization and block strategy initialization, both of which have nearly perfect load balancing, is due to the simplicity of the block strategy.

The variations in the primordial phase execution times are also attributable to load imbalance, although the load imbalance is not caused entirely by the number of solutions assigned to each processor. The degree of compatibility among solutions assigned to the processor also contributes significantly to execution time in the primordial phase. Where the degree of compatibility is high, as with the indexed, modified indexed, and block strategies, the second mate for each tournament is found relatively quickly. Conversely, where the degree of compatibility is low, as with the interleaved strategy, additional time is required to find a compatible mate. The overall effect is that the interleaved strategy results in significantly higher execution time in the primordial phase than the other strategies.

The indexed, modified indexed, and block distribution strategies all result in "super-linear speedup" of the primordial phase. The modified indexed and block distribution strategies also result in "super-linear speedup" of the initialization, primordial phase, and conversion operations together. The presence of "super-linear speedup" is misleading in that the parallel algorithm is not completely functionally equivalent to the sequential algorithm. The modifications introduced in the parallelization account for part of the speedup. Application of the same modifications to the sequential algorithm should result in reduced execution time.

5.4 Summary.

The execution time of the MGA is dominated by the primordial phase, indicating that significant speedup requires parallelization of the tournament selection algorithm. The algorithm is inherently sequential due to the property that any tournament may involve any member of the population. Parallel algorithms exist which approximate the behavior of the sequential tournament selection algorithm. The initial population may be distributed in a number of ways, including the "indexed," "modified indexed," "interleaved," and "block" distribution strategies. The indexed and modified indexed strategies allocate solutions to processors based upon the solution's first defined locus, resulting in uneven distribution. The interleaved and block distribution strategies result in even distributions.

Primordial phase execution time is a function of the number of solutions in the subpopulation, the probablity that two randomly selected solutions are compatible, and the shuffle size. The distribution strategy affects the number of solutions in the subpopulation and the probability that two solutions are compatible. Experiments comparing the solution quality and execution time of the four distribution strategies when applied to the fully deceptive binary function show that the distribution strategy does not have a significant effect on solution quality. Results indicate that the distribution strategy does have a significant effect on primordial phase execution time. The indexed, modified indexed, and block distribution strategies result in "super-linear speedup" in the primordial phase, indicating that the sequential algorithm can be made more efficient by partitioning the population.

VI. Communication in Parallel Genetic Algorithms.

One of the objectives of this thesis is to investigate methods of implementing simple genetic algorithms on parallel architectures. As discussed in Section 2.8, reported parallel implementations of genetic algorithms have exploited the high degree of data parallelism present in genetic algorithms by partitioning the population among the processors(8, 7, 45, 30, 46). Partitioning the population requires design decisions regarding the implementation of the selection and crossover operators.

Selection may be performed locally or globally. Local selection treats each subpopulation independently. Independent selection eliminates the communication overhead and synchronization requirements associated with global selection. On the other hand, local selection introduces selection bias, which can be avoided in global selection. Section 6.1 examines local selection, sequential global selection, and parallel global selection in the context of Baker's metrics and other considerations specific to parallel implementations(3, 14).

Likewise, crossover may be performed either locally or globally. Global crossover allows subpopulations to benefit from information gained from the search processes conducted by other subpopulations, but only at the expense of unacceptable communication overhead. As an approximation to global crossover, subpopulations may "share" highly fit solutions with other subpopulations. Like global crossover, "solution sharing" allows subpopulations to benefit from information obtained in other subpopulations' search processes. In contrast to global crossover, solution sharing incurs only modest communication overhead. However, solution sharing artificially increases the number of copies of each communicated solution, and as such may lead to premature convergence. Premature convergence in genetic algorithms occurs when populations become dominated by similar strings representing locally optimal solutions early in the search process(15).

This chapter presents three solution sharing strategies (Section 6.2), including a non-sharing strategy, an unconditional sharing strategy(14), and a conditional strategy(35). The non-sharing strategy performs no solution sharing. The unconditional sharing strategy communicates solutions regardless of whether or not they have previously communicated. The conditional sharing strategy only communicates solutions which are better than

previously communicated solutions. The remainder of the chapter describes experiments comparing the effects on premature convergence of each of the selection and solution sharing strategies (Section 6.3) and presents the results of the experiments (Section 6.4). These experiments extend Dymek's work(14) by examining the unconditional sharing strategy in conjunction with the parallel global selection strategy, and by adding the conditional sharing strategy experiments. The results of the experiments are presented in Section 6.4.

6.1 Selection Strategies

Baker identifies three metrics by which to compare selection algorithms which are oriented towards maintaining the validity of the Schema Theorem and reducing execution time(3). Following Baker, let $f(i)$ be the actual number of copies allocated to individual i and $ev(i)$ the expected number of copies assuming perfect sampling. "Perfect sampling" is defined such that

$$ev(i) = \frac{\mu_i N}{\sum_{j=1}^{N} \mu_j}, \tag{15}$$

where μ_i is the fitness of individual i and N is the population size. An individual's "bias," which was first studied by Brindle(5), is the absolute value of the difference between its "actual" sampling probability and its expected sampling probability. Assuming that the "actual" sampling probability is well approximated by the mean sampling rate $\overline{f(i)}/N$, observed over some large number of independent executions of the selection algorithm, and noting that the expected sampling probability is $ev(i)/N$, the bias of individual i is given by

$$B(i) = \left| \frac{\overline{f(i)} - ev(i)}{N} \right|. \tag{16}$$

"Zero bias" is present when the mean number of copies allocated to an individual is equal to the expected number of copies. "Spread" is the range of possible values for $f(i)$. "Minimum Spread" is the smallest spread which theoretically permits zero bias, implying that

$$f(i) \in \{\lfloor ev(i) \rfloor, \lceil ev(i) \rceil\}. \tag{17}$$

Baker's third metric, efficiency, is the time complexity of the selection algorithm. Bias, spread, and efficiency provide a useful and complete framework for the evaluation of sequential selection algorithms and a starting point for the evaluation of parallel selection algorithms.

Parallel selection algorithms may exhibit bias and spread with respect to either the local or global population. Local and global bias are given by Equation 16 where $ev(i)$ is calculated based upon the mean fitness of the local population or the global population respectively. Likewise, local and global spread are both given by Equation 17, with appropriate values for $ev(i)$. In order to ensure the applicability of the Schema Theorem, a parallel selection algorithm must exhibit zero global bias and minimum global spread. Additional issues concerning parallel selection algorithms include load balancing, synchronization requirements, and communication requirements.

As discussed in Section 2.3, the most common implementation of selection is stochastic sampling with replacement (SSR). SSR has zero bias, but also has unlimited spread and an algorithmic time complexity of $O(N^2)$, where N is the population size[1]. Baker proposes a selection algorithm, shown at Figure 43, called "Stochastic Universal Sampling" (SUS) which is bias-free, has minimum spread, and has an algorithmic time complexity of $O(N)$(3:16-17,19). The remainder of this section presents three parallel selection algorithms, each of which is based on SUS. Each algorithm is discussed in the context of Baker's metrics and the above parallel performance measures. All three strategies begin by evenly distributing N/m solutions to each of the m processors.

The first strategy, local selection, performs SUS for each subpopulation independently. Local selection allocates copies to solutions based upon their fitness relative to the average fitness of solutions in the same subpopulation. Thus, it allocates

$$f(i) = \frac{\mu_i N}{m \sum_{k=1}^{N/m} \mu_k} \tag{18}$$

[1] $O(N log N)$ using a B-tree.

```
1. Compute total fitness
2. ptr ← Rand()
3. sum ← 0, i ← 0, k ← 0
4. while i < N
    (a) expected ← fitness[i] × N / total fitness
    (b) sum ← sum + expected
    (c) while ptr < sum
          i. sample[k] ← i
          ii. k ← k + 1
          iii. ptr ← ptr + 1
    (d) i ← i + 1
```

Figure 43. Baker's Stochastic Universal Sampling (SUS) Algorithm

```
1. Compute total fitness
2. ptr ← Rand()
3. sum ← 0, i ← 0, k ← 0
4. while i < N/m
    (a) expected ← fitness[i] × (N/m) / total fitness
    (b) sum ← sum + expected
    (c) while ptr < sum
          i. sample[k] ← i
          ii. k ← k + 1
          iii. ptr ← ptr + 1
    (d) i ← i + 1
```

Figure 44. Local Selection Algorithm

copies to each solution in the ith subpopulation. Substitution into Equation 16 and using the global average fitness to calculate $ev(i)$ shows that local selection results in global bias

$$\begin{aligned} B(i) &= \left| \frac{\mu_i}{m \sum_{k=1}^{N/m} \mu_k} - \frac{\mu_i}{\sum_{k=1}^{N} \mu_k} \right| \\ &= \mu_i \left| \frac{1}{m \sum_{k=1}^{N/m} \mu_k} - \frac{1}{\sum_{k=1}^{N} \mu_k} \right| \end{aligned} \tag{19}$$

towards individuals allocated to subpopulations for which the average fitness is below the overall population average fitness.

Local selection does not vary the local population sizes, and each subpopulation contains N/m solutions (Figure 44). The value of the **sum** variable at the completion of the algorithm is equal to the total of the values taken on by the **expected** variable over

the N/m iterations of the outer loop:

$$\sum_{i=1}^{N/m} \frac{\mu_i \times N/m}{\sum_{j=1}^{N/m} \mu_j} = \frac{N}{m}.$$

Some iterations of the outer loop result in multiple executions of the inner loop, while others do not result in any. Every execution of the inner loop increments the `ptr` variable. At the completion of the algorithm,

$$\texttt{ptr} = \lceil \texttt{sum} \rceil = \left\lceil \frac{N}{m} \right\rceil.$$

Therefore, the *total* number of times which the inner loop executes is $\lceil \frac{N}{m} \rceil$, so that local selection is of $O(N/m)$ algorithmic time complexity on each processor. Because each processor simultaneously executes exactly the same number of operations, the overall complexity is also $O(N/m)$ and local selection exhibits perfect load balancing. Local selection has no synchronization or communication requirements.

The second strategy, sequential global selection (or simply "global selection") performs SUS on the entire population. The SUS algorithm is modified (Figure 45) to include communication between processors during the global fitness calculation and after each subpopulation is selected. Global selection results in zero global bias, minimum global spread, and low communication time. Because global selection has zero bias, the number of copies each solution receives is given by Equation 15. The total number of solutions in subpopulation i in generation $t+1$ is then given by

$$\begin{aligned} N_i(t+1) &= \sum_{j=1}^{N_i(t)} \frac{\mu_j N}{\sum_{k=1}^{N} \mu_k} \\ &= N \frac{\sum_{j=1}^{N_i(t)} \mu_j}{\sum_{k=1}^{N} \mu_k}. \end{aligned} \tag{20}$$

which predicts that a subpopulation's size in the next generation is proportional to its contribution to the total fitness in the current generation. Thus, better solutions lead to larger subpopulations.

Goldberg presents theoretical and experimental arguments for the existence of optimal population sizes for both serial and parallel implementations of genetic algorithms(22). The optimal size depends upon a number of factors, including string length, cardinality, and degree of parallelization. For population sizes below the optimal, insufficient schemata are represented and the genetic algorithm tends to converge prematurely. Population sizes above the optimal increase execution time without significantly increasing the number of schemata represented. Because better solutions in sequential global selection lead to larger subpopulations, it is reasonable to expect that load balancing becomes an issue when the subpopulation sizes are below the optimal.

Applying reasoning similar to that used in the analysis of the local selection algorithm, at the completion of the sequential global selection algorithm,

$$\texttt{global\_sum} = \lceil N \rceil = N.$$

Thus, the inner loop of Figure 45 is executed a *total* of N times by all the processors. Each processor i executes the loop $N_i(t)$ times. Due to the synchronization requirement following the selection of each subpopulation and prior to the selection of the next subpopulation, the executions of the inner loop do not overlap. Thus, the global selection algorithm has $O(N)$ time complexity.

The third strategy, parallel global selection, which is due to Dymek, relaxes the requirement for perfect adherence to theoretical selection rates(14). The global selection algorithm is modified to eliminate the communication step following subpopulation selections, and therefore the associated synchronization requirement. The parallel global selection algorithm is shown at Figure 46. Using the roulette wheel analogy, if Baker's original algorithm is like spinning one large wheel, the parallel version is similar to spinning one smaller wheel on each processor (See Figure 47). It results in zero bias, minimum spread, and low communication time. Again, the inner loop of Figure 46 is executed a total of N times, where each processor i executes it $N_i(t)$ times. As is the case with the local selection algorithm, execution on separate processors overlaps, so that the total execution time is determined by the largest subpopulation size. Thus, the parallel global selection algorithm

1. Compute total_fitness and global_total_fitness
2. if mynode() = 0 then global_ptr ← Rand() else crecv(global_ptr)
3. $\forall i$: $0 < i \leq N$ :: global_sample[i] ← −1,
4. if mynode() = 0 then global_sum = 0 else crecv(global_sum)
5. expected_sum ← 0, i ← $\sum_{i=0}^{mynode()-1} N_i(t)$, m ← 0
6. while i < $\sum_{i=0}^{mynode()} N_i(t)$
   (a) global_expected ← fitness[i] × N / global_total_fitness
   (b) expected_sum ← expected_sum + global_expected
   (c) global_sum ← global_sum + global_expected
   (d) while global_ptr < global_sum
      i. global_sample[m] ← i
      ii. m ← m + 1
      iii. global_ptr ← global_ptr + 1
   (e) i ← i + 1
7. if mynode() ≠ numnodes() - 1 csend(global_ptr), csend(global_sum)
8. Determine new local population size

Figure 45. Global Selection Algorithm

is of $O(A)$ algorithmic time complexity where A is the largest subpopulation size(14). The computation of the global total fitness and global population size require synchronization of the processors prior to each application of the selection operator. As with sequential global selection, load balancing is likely to become an issue when subpopulation sizes are below the optimal.

6.2 Solution Sharing Strategies

While global selection is a reasonable option, global crossover is not. A parallel genetic algorithm implemented on m processors using global crossover with a probability of crossover p_c, would on average communicate a fraction $p_c(m-1)/m$ of the solutions in the population every generation. Using conservative values of $p_c = .6$ and $m = 8$, global crossover would require communicating an average of 53% of the solutions every generation. Increased probabilities of crossover or numbers of processors result in even higher communication requirements. Current parallel architectures are such that inter-processor communication is substantially slower than computation. Were this not the case, global crossover would be a viable option.

1. Compute total_fitness and global_total_fitness
2. ptr ← Rand()
3. ∀i: 0 < i ≤ N :: global_sample[i] ← −1,
4. sum = 0, i ← $\sum_{i=0}^{mynode()-1} N_i(t)$, m ← 0
5. while i < $\sum_{i=0}^{mynode()} N_i(t)$
   (a) global_expected ← fitness[i] × N / global_total_fitness
   (b) sum ← sum + global_expected
   (c) while ptr < sum
      i. global_sample[m] ← i
      ii. m ← m + 1
      iii. ptr ← ptr + 1
   (d) i ← i + 1
6. Determine new local population size and new global population size N

Figure 46. Parallel Global Selection Algorithm

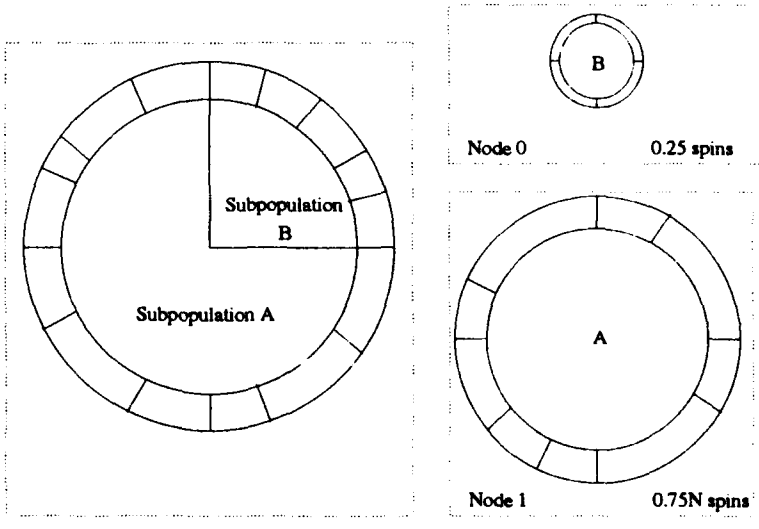

Figure 47. Parallelizing Baker's Algorithm

Whether a parallel genetic algorithm implementation uses local, global, or parallel global selection, it is not functionally equivalent to a sequential implementation, if crossover is performed locally. The absence of global crossover reduces the efficiency of schema processing within the genetic algorithm, because new solutions are formed only from portions of solutions present in the same subpopulation.

Kommu presents expressions for trial allocation in parallel genetic algorithms with and without communication for a two armed bandit problem(36). A solution to a k-armed bandit problem attempts to maximize the total payoff obtained in sequential selections from $k \geq 2$ arms, each of which has an unknown payoff distribution. Kommu's expression for trial allocation to a schema ζ in generation $t+1$ in a genetic algorithm implemented on P processors with no communication predicts

$$\begin{aligned} m(\zeta, t+1) &= \sum_{i=1}^{P} \left[\frac{f_\zeta^i}{\frac{f_\zeta^i + f_{\zeta'}^i}{\rho(t)}} \right] \\ &= \rho(t) \sum_{i=1}^{P} \left[\frac{f_\zeta^i}{f_\zeta^i + f_{\zeta'}^i} \right], \end{aligned} \tag{21}$$

where f_ζ^i is the combined fitness of the samples allocated to ζ in the ith subpopulation in iteration t, $f_{\zeta'}^i$ is the number allocated to the other schema ζ', and $\rho(t)$ is the size of each subpopulation, which is assumed to be constant. The equivalent expression for trial allocation in a parallel genetic algorithm with communication predicts[2]

$$\begin{aligned} m(\zeta, t+1) &= \left[\frac{\sum_{i=1}^{P} f_\zeta^i}{\frac{\sum_{i=1}^{P} f_\zeta^i + \sum_{i=1}^{P} f_{\zeta'}^i}{P \times \rho(t)}} \right] \\ &= P\rho(t) \left[\frac{\sum_{i=1}^{P} f_\zeta^i}{\sum_{i=1}^{P} f_\zeta^i + \sum_{i=1}^{P} f_{\zeta'}^i} \right]. \end{aligned} \tag{22}$$

The latter actually predicts the trial allocation assuming perfect communication, i.e. global crossover(34). Solution sharing approximates the effects of global crossover by communicating the best solution(s) present in a subpopulation to other subpopulations. In order

[2]The limits of summation in the denominator have been changed to correct a presumed typographical error. The limits as published in *Proceedings of the 1992 International Conference on Parallel Processing* ranged from i to P for both summations.

to predict the effect of solution sharing on trial allocation, it is convenient to assume that the communication of a single solution containing the schema ζ results in an increase in f_ζ^i

$$f_\zeta^{i*} = \frac{m(\zeta,t)+1}{m(\zeta,t)} f_\zeta^i, \tag{23}$$

and a corresponding decrease in $f_{\zeta'}^i$

$$f_{\zeta'}^{i*} = \frac{m(\zeta',t)-1}{m(\zeta',t)} f_{\zeta'}^i, \tag{24}$$

so that the trial allocation in the next generation may be approximated as

$$\begin{aligned} m(\zeta,t+1) &\approx P\rho(t)\left[\frac{\sum_{i=1}^P f_\zeta^{i*}}{\sum_{i=1}^P f_\zeta^{i*} + \sum_{i=1}^P f_{\zeta'}^{i*}}\right] \\ &\approx P\rho(t)\left[\frac{\sum_{i=1}^P \frac{m(\zeta,t)+1}{m(\zeta,t)} f_\zeta^i}{\sum_{i=1}^P \frac{m(\zeta,t)+1}{m(\zeta,t)} f_\zeta^i + \sum_{i=1}^P \frac{m(\zeta',t)-1}{m(\zeta',t)} f_{\zeta'}^i}\right] \\ &\approx P\rho(t)\left[\frac{(m(\zeta,t)+1)\sum_{i=1}^P f_\zeta^i}{(m(\zeta,t)+1)\sum_{i=1}^P f_\zeta^i + (m(\zeta',t)-1)\sum_{i=1}^P f_{\zeta'}^i}\right]. \end{aligned} \tag{25}$$

Assuming that $m(\zeta,t) \approx m(\zeta',t)$ in early generations, and observing that in large populations $m(\zeta,t) \gg 1$, Equation 25 reduces to Equation 22, so that the approximation to global crossover is good. In very small populations, on the other hand, $m(\zeta,t) \approx 1$, so that the contribution of ζ' approaches zero, and Equation 25 predicts rapid convergence.

Solution sharing thus allows receiving subpopulations to benefit from information obtained in the search processes conducted by transmitting subpopulations. On the other hand, sharing creates a disproportionate number of copies of the communicated solutions in the global population, which may lead to premature convergence, especially in small populations. The remainder of this section presents three solution sharing strategies with varying degrees of communication overhead and anticipated effects on premature convergence, which are compared experimentally in Section 6.3.

The first solution sharing strategy, "No sharing," performs no solution sharing. The no sharing strategy does not result in disproportionate copies of solutions, nor does it require communication overhead. It also does not allow subpopulations to share information,

thereby reducing the efficiency of schema processing. Trial allocation for this strategy is predicted by Equation 21, so that premature convergence is not anticipated.

The second strategy, "Sharing," periodically broadcasts a single solution from each subpopulation. Other implementations of parallel genetic algorithms have used periodic communication(58). The number of generations in between communication is referred to as the *epoch* size. This study uses an epoch size of 5 generations. Received solutions replace less fit solutions in the receiving population. This strategy allows subpopulations to share information, but allocates extra copies to the communicated solutions and involves a small amount of communication overhead. Trial allocation for this strategy is approximated by Equation 25, so that premature convergence in small populations is anticipated.

The third strategy, "Conditional sharing," is identical to the second, except that solutions are communicated only when they represent an improvement over the previous solution communicated by that node(35). This strategy allows subpopulations to share the most valuable information, while minimizing the extra copies allocated due to sharing. The implementation of conditional sharing used in this study involves even greater communication overhead than the "Sharing" strategy, although in a more efficient implementation this would not be the case. Trial allocation for this strategy is approximated by Equation 25 for generations in which solutions are communicated, and by Equation 21 for generations in which no solutions are communicated. Because convergence itself tends to reduce the rate at which new solutions are explored, this strategy decreases the rate of solution sharing when the subpopulations begin to converge. The anticipated effect is an overall improvement in solution quality relative to the first strategy, and a reduction in the tendency towards premature convergence relative to the second strategy.

6.3 Premature Convergence Experiments.

As discussed in Section 4.4, Rosenbrock's Saddle is an established test of genetic algorithm performance(13, 21, 45). It is also routinely used as a standard against which to judge the performance of gradient-based search using the penalty function approach. Dymek reports that the function belongs to a class of functions associated with control sys-

tems used in air-to-air missile guidance, tracking, and oscillation control(14). Rosenbrock's saddle is used as the test problem in this study because

- premature convergence was observed using a hypercube implementation of a genetic algorithm(45:156-158),
- premature convergence could not be attributed to population size alone(45:156-158),
- premature convergence was not alleviated using a strategy which prevents similar solutions from mating(15:115-122),
- the function is not GA-hard,
- local selection is cited as a possible cause of premature convergence(45:156), and
- previous work examining premature convergence in parallel genetic algorithms focused on Rosenbrock's saddle(14).

Each of the three selection strategies is tested in conjunction with each of the three solution sharing strategies. For each combination, global population size is varied from 80 to 3200 to examine performance in both small and large populations. The number of generations is fixed at 200. In order to reduce the effects of noise due to dependence on the initial random number seed, the results for each test case are averaged over 40 executions using different random seeds. The seeds are themselves randomly generated in such a manner as to ensure that the initial populations of each execution may be assumed to be independent(14). Identical random seeds are used across test cases to eliminate variance due to dependence on the random number sequence. The encoding scheme and evaluation function are the same as those described in Section 4.4.

6.4 Results.

This section presents the results of the communication strategy experiments. The results are specific to the optimization of Rosenbrock's Saddle, and are not necessarily valid for other problems. None of the data are tested for statistical significance, so that the results should be interpreted only as trends.

Two Letter Designator	Selection Strategy	Sharing Strategy
LN	(L)ocal	(N)onsharing
LS	(L)ocal	Unconditional (S)haring
LC	(L)ocal	(C)onditional Sharing
GN	Sequential (G)lobal	(N)onsharing
GS	Sequential (G)lobal	Unconditional (S)haring
GC	Sequential (G)lobal	(C)onditional Sharing
PN	(P)arallel Global	(N)onsharing
PS	(P)arallel Global	Unconditional (S)haring
PC	(P)arallel Global	(C)onditional Sharing

Table 38. Two Letter Designators for Communication Strategies

Each combination of selection strategy and sharing strategy is given a two letter designator, selected to maintain consistency with Dymek's designators, as shown in Table 38. Population sizes are divided into two groups. "Small" population sizes are those less than or equal to 320, while those greater than 320 are "large." The "effectiveness statistic" E of a strategy, which gives a rough indication of the effectiveness of the search process in the last half of each execution, is arbitrarily defined as

$$E = \frac{F(100) - F(200)}{F(100)}, \tag{26}$$

where $F(t)$ is the best fitness of any solution in generation t for the strategy in question. An effectiveness statistic near 0 indicates premature convergence, while an effectiveness statistic near 1 indicates continued effective search. The effectiveness statistics of each of the strategies for each population size is shown in Appendix C. The mean effectiveness statistics for each strategy taken over all small populations and over all large populations are shown at Tables 39 and 40, respectively. As expected, the data indicate that small populations are more prone to premature convergence than large populations. In particular, unconditional solution sharing in small populations contributes to premature convergence. Strategies using conditional solution sharing are slightly less susceptible than those using no sharing. Global selection in small populations appears to contribute to premature convergence, and parallel global selection appears to contribute more than sequential global

Selection Strategy	Effectiveness Statistic	
	Mean	Standard Deviation
LN	0.256	0.135
LS	0.021	0.038
LC	0.372	0.160
GN	0.208	0.180
GS	0.055	0.087
GC	0.300	0.229
PN	0.177	0.174
PS	0.004	0.007
PC	0.189	0.121

Table 39. Effectiveness Statistics – Small Populations

Selection Strategy	Effectiveness Statistic	
	Mean	Standard Deviation
LN	0.787	0.054
LS	0.577	0.354
LC	0.779	0.066
GN	0.706	0.073
GS	0.535	0.273
GC	0.804	0.070
PN	0.514	0.107
PS	0.288	0.098
PC	0.599	0.073

Table 40. Effectiveness Statistics – Large Populations

selection. Large populations are not immune to premature convergence. In particular unconditional sharing and parallel global selection both contribute to premature convergence.

Raw data displaying solution quality as a function of generation for each population size and communication strategy is given in Volume II. Section C.1 of this thesis shows execution time as a function of population size.

In small population sizes, solution sharing strategies appear to have a more significant effect on solution quality than do selection strategies. As expected, conditional sharing strategies consistently perform better than non-sharing strategies, which in turn perform better than unconditional sharing strategies. This is consistent with the behavior predicted by Equations 21 and 25. Parallel global selection tends toward lower solution quality than both local selection and sequential global selection, due to the selection errors induced in parallelizing the global selection algorithm.

In large population sizes, selection strategy has a significant effect on solution quality. Local selection strategies, which converge quickly on locally optimally solutions, obtain better solutions than sequential global selection strategies overall, many of which do not converge prior to the completion of the experiment. Sequential global selection strategies may produce better solutions in longer experiments. One exception to local selection's superiority is that LC and GC obtain nearly equal solution quality. Parallel selection strategies do not perform as well as other strategies. In many cases, unconditional sharing results in better overall solutions than either the non-sharing or the conditional sharing strategy, although differences in convergence characteristics indicate that the latter strategies might overtake the former in longer executions.

Execution times for each of the communication strategies are shown in Appendix C, grouped into small and large population sizes.

In small population sizes, the correlation between execution time and population size is weak for PN, and PC. The correlation is strong for all three local selection strategies, and almost as strong for GN, GS, GC, and PS. Execution times for the local selection strategies grow linearly with population size, but reflect the communication overhead involved with the solution sharing strategies. In the smallest population sizes, GS (PS) has

longer execution times than GN and GC (PN and PC). In larger "small" populations this is not the case.

The correlation between execution time and population size is stronger in large populations than in small populations. Local selection results in nearly identical execution times for all sharing strategies, which grow linearly with population size. The execution times for GN and GC are roughly 10% higher than those for GS. Also, the execution times for PN and PC are roughly 10% higher than those for PS. Unconditional sharing apparently maintains relatively even population sizes even when global selection is used. Execution times for global selection strategies are less well behaved than those for local selection strategies, but still grow roughly linearly with population size. The execution times for parallel global selection are 30-40% higher than those for local selection, while those for sequential global selection are 80-100% higher. The higher execution times can be attributed to load imbalance.

6.5 Summary.

Baker's proposes bias, spread, and efficiency as metrics for comparison of genetic algorithm selection algorithms and presents the Stochastic Universal Sampling (SUS) algorithm. SUS exhibits zero bias, minimum spread, and is of $O(N)$ time complexity. In order to apply the bias and spread metrics meaningfully to parallel selection algorithms, they must be defined with respect to the global population. Parallel selection involves several other issues, including load balancing, synchronization requirements, and communication overhead.

Three parallel versions of the SUS algorithm are local SUS, sequential global SUS, and parallel global SUS. Local SUS has the advantages of $O(N/m)$ time complexity, perfect load balancing, and no synchronization or communication requirements. It has the disadvantages of non-zero global bias and non-minimum spread. Sequential global SUS has zero global bias and minimum global spread, but is of $O(N)$ time complexity, results in load imbalance, and requires synchronization and some communication. Parallel global SUS approximates sequential global SUS while eliminating the synchronization require-

ments. As a result, it has near zero global bias and near minimum spread, is of $O(A)$ time complexity, and results in load imbalance.

Solution sharing is an approximation to global crossover with greatly reduced communication requirements. The solutions communicated are allocated more copies than predicted by the Schema Theorem, potentially leading to premature convergence. Conditional sharing reduces the tendency to premature convergence by only communicating solutions which are better than previously communicated solutions.

Experiments comparing the effects on premature convergence of each of the selection and solution sharing strategies for various population sizes are summarized in Section 6.4.

VII. Conclusions.

7.1 Generalization of the Messy Genetic Algorithm.

The generalized messy genetic algorithm is demonstrated to be capable of consistently finding near optimal solutions to the Traveling Salesman Problem. In some experiments, it finds the globally optimal solution. This is the first application of a messy genetic algorithm to a combinatoric optimization problem. The generalized MGA is shown experimentally to perform better than AFIT's original MGA implementation(14), the GENESIS simple genetic algorithm, and the permutation version of GENESIS(27).

The generalized version is also demonstrated to be able to consistently find near-optimal solutions to an order-3 fully deceptive binary function optimization problem(14), which is provably difficult for simple genetic algorithms. In many cases, the generalized MGA finds the optimal solution. The generalized MGA is shown experimentally to perform better than the GENESIS simple genetic algorithm(27), although AFIT's original MGA implementation(14) obtains slightly better solutions.

Finally, the generalized MGA is demonstrated to be capable of consistently finding near optimal solutions to DeJong function f2(13), also known as Rosenbrock's Saddle. The generalized MGA is shown experimentally to perform better than AFIT's original MGA implementation(14), but not as well as the GENESIS simple genetic algorithm(27).

7.2 Parallelization of the MGA Primordial Phase.

AFIT's hypercube MGA implementation(14), which interleaves the initial population members among the processors, is compared to a block distribution strategy and two other strategies. The "indexed" and "modified indexed" allocate population members to processors based upon an index. A solution's index is the first string position in which the solution is defined. The "indexed" strategy uses standard interleaving based on the index, while the "modified indexed" reverses the order in which processors receive solutions in all passes following the initial one. The "modified indexed" strategy achieves a more even distribution of population members than the "indexed" strategy. Choice of initial

population distribution strategy is not found to have a statistically significant effect on solution quality.

Even though the interleaving strategy allocates the members of the initial population to the processors as evenly as possible, it is found to require significantly higher execution times for the primordial phase than the other distribution strategies. This appears to be due to the extra time spent searching for compatible mates with which to perform tournaments. The "modified indexed" strategy results in the lowest primordial phase execution times.

Choice of distribution strategy also has a significant effect on the execution time required to merge the subpopulations and convert the data structures prior to the juxtapositional phase.

7.3 Communication in Parallel Genetic Algorithms.

A total of nine communication strategies are examined, representing all possible combinations of three selection strategies and three solution sharing strategies. An existing theoretical model of trial allocation for the Stochastic Universal Sampling (SUS) selection algorithm(3) is extended to predict the behavior of three parallel implementations. Likewise, an existing theoretical model of schema growth for solution sharing(36) is refined and extended to predict the behavior of three solution sharing strategies.

Experiments comparing the solution quality, execution time, and convergence characteristics of each of the nine communication strategies for 200 generations in an 8-node iPSC/2 implementation solving DeJong function f2(13) lead to a number of observations. In these experiments,

- small populations are more prone to premature convergence than large populations,
- unconditional solution sharing contributes significantly to premature convergence in small populations,
- conditional solution sharing reduces the tendency to premature convergence in small populations,

- sequential global SUS leads to premature convergence in small populations,
- parallel global SUS increases the tendency to premature convergence in small populations,
- unconditional solution sharing and parallel global SUS both lead to premature convergence in large populations,
- conditional solution sharing in small populations results in better overall solution quality than no solution sharing or unconditional solution sharing,
- local SUS in large populations results in better solution quality in generation 200,
- parallel global SUS in small populations results in unpredictable execution times,
- unconditional sharing in very small populations results in longer execution times,
- unconditional solution sharing in moderate and large populations results in lower execution times, except in conjunction with local SUS,
- sequential global SUS in large populations results in execution times approximately 80 - 100% longer than local SUS, and
- parallel global SUS in large populations results in execution times approximately 30 - 40% longer than local SUS

Overall, for large population sizes, local selection with unconditional sharing results in the best solution quality in these experiments. Other strategies may produce better solutions in longer experiments.

VIII. Recommendations.

8.1 Further Investigation of MGA Performance.

The generalized MGA does not perform as well as AFIT's original MGA on the fully deceptive binary function optimization problem. Investigation of the reasons for the poorer performance may lead to a better understanding of the MGA's behavior and improved performance. Also, the generalized MGA performs significantly better than the permutation GA on the 12-city TSP. Experiments in which the string length is much larger than the block size are necessary in order to determine whether or not the results are extensible to larger problem sizes. Finally, the generalized messy genetic algorithm does not perform as well as the simple genetic algorithm on DeJong function f2(13). This may be attributable to nonuniform building block size(24). Incorporation of Goldberg's recommendations for null bits with tie breaking might lead to improved understanding and performance.

8.2 Reduction of the MGA Initialization Phase Memory Requirement.

The large memory requirement of the MGA's partially enumerative initialization restricts the applicability of the MGA significantly. Tournament selection imposes an ordering on each set of solutions based upon their fitnesses. The number of copies allocated to a solution depends only on its position in the ordering, not on its actual fitness. Therefore, an initialization algorithm which determines the distribution of solutions in the initial juxtapositional phase prior to the enumeration of the solutions is possible. The degree of compatibility among solutions in the population must be considered. A C fragment for an algorithm which computes an expected distribution is shown at Figure 48. The algorithm assumes that no two solutions share the same fitness. The design of an algorithm which does not depend on this assumption is critical to accurately modeling tournament selection.

```
size = 0;
for (i = 0; i < generations; i++)
   winners[i] = 0;

for (j = 0; size < pop_size[generations]; j++) {
   m[j] = 1.0;
   for (i = 0; i < generations; i++) {

      tournaments =
         (double) shuffle_size * (double) m[j] *
                  ((double) pop_size[i + 1] / (double) pop_size[i]);

      survival_rate =
         (double) (pop_size[i] - winners[i]) / (double) pop_size[i];

      m[j] = (double) tournaments * survival_rate;

      extra = m[j] - (int) m[j];

      winners[i] += m[j] = (Rand() < extra) + (int) m[j];
  }

   size += m[j];
}

m[j - 1] -= (double) (size - pop_size[generations]);
```

Figure 48. Stochastic Remainder Multi-Generational Tournament Selection C Fragment

8.3 Application of the MGA to Deceptive Ordering Problems.

As discussed in Section 2.6, Kargupta presents fully deceptive absolute ordering and fully deceptive relative ordering problems(31). Application of the MGA to these problems is important to a complete understanding of its capabilities with regard to combinatoric optimization problems. AFIT's current MGA implementations cannot be applied to these problems due to memory constraints.

8.4 Application of the MGA to the Conformational Analysis Problem.

It is important to demonstrate the applicability of messy genetic algorithms to problems of practical interest to the Air Force. A specific problem for which preliminary work is complete is the conformational analysis of polypeptides. The conformational analysis problem is representative of a class of computationally difficult problems, and is also of interest due to its importance in the design of materials with specific non-linear optical properties. The 3-dimensional structure of a large molecule, which is referred to as its *conformation*, is the primary influence on the electrical and mechanical properties of the material(33). Predicting the conformation based solely on knowledge of the molecule's covalent bonding is non-trivial at best, and in general is not possible. Knowledge of the hydrogen bonding is usually necessary as well.

The polypeptide conformational analysis problem is often solved via minimization of the energy function, which has been accomplished using many optimization techniques. Genetic algorithms (GA) have been applied to the related problem of DNA conformational analysis by maximizing the correlation of J-coupling data obtained from X-ray diffraction. The energy minimization, Monte Carlo simulation, and simulated annealing techniques search for molecular conformations which minimize the energy function of the molecule(39:298-308). The energy function is modeled empirically, and usually consists of contributions due to bond stretch, bond angle deformation, hindered torsional angle deformation, and non-bonded interactions, as shown previously in Equation 6.

An attractive application of the messy genetic algorithm is to the conformational analysis of [Met]-Enkephalin, a polypeptide consisting of the five-residue sequence Tyr-

Gly-Gly-Phe-Met, which was used by Nayeem, Vila, and Scheraga to compare simulated annealing and Monte Carlo simulation(42). Several simplifying assumptions are in order to reduce computational complexity:

- Rigid covalent geometry of the residues. Taking the dihedral angles ϕ, ψ, ω, and χ as the only independent variables reduces the number of independent variables to a manageable number, and allows direct comparison to the results of Nayeem, et. al.(42).
- United-atom model potential energy functions. Non-polar hydrogen atoms are not represented, and the carbon atoms to which they are bonded are modified to account for their effects.

The remainder of this section proposes a design for this application. The encoding scheme for the energy function is based upon a string of 24 genes and a genic alphabet of cardinality 3600. Each gene corresponds to one of the free bond or torsion angles, each of which can take on any value between 0 and 360 degrees, in 0.1 degree increments. The encoding scheme uses a string length of 24, with each gene corresponding to a single independent variable.

The fitness function for the application is a C translation of that portion of the MM2 molecular dynamics code which performs energy function evaluation. MM2 is well established in the computational chemistry community. It supports use of the Z-matrix representation of a molecule, which allows direct evaluation of the angles represented by the genes in the MGA strings. It includes contributions due to bond stretch, bond angle deformation, torsion angle hindered rotation, and non-bonded interactions. The evaluation function takes as input a fully specified solution in MGA representation. It first decodes the input arguments, using the map table established by the initialization function to complete the specification of the molecule. It then computes the total energy of the molecule, which it returns as the fitness of the solution.

The domain initialization function defines the cardinality of the genic alphabet and initializes the molecular data. It obtains the molecular data from a file containing a Z-matrix representation of the molecule under study. The Z-matrix representation specifies

a bond length, a bond angle, and a torsion angle for each atom, except in cases where fewer constraints are required to unambiguously define the position of an atom. It also designates which of the variables are free. The initialization function creates a mapping table which uniquely identifies each free variable with a string position.

The overlay function takes as input a partial solution and a template, both in MGA representation. It produces a fully specified solution by taking genes corresponding to unspecified loci, first from the partial solution, then from the template.

AFIT's current MGA implementations are not capable of supporting genic alphabets of cardinality more than 255. Thus, implementation of this design necessitates modification of the MGA to support larger genic alphabets. Also, the proposed large genic alphabet implies a very large number of solutions in the initial population. Therefore, this application also necessitates the reduction of the memory requirement described above.

8.5 Extension of Parallelization Experiments.

The results of this study are, strictly speaking, valid only for the particular problems examined. In order to better assess their implications in general, application of the parallel messy genetic algorithm and parallel genetic algorithm to other problems is required. Furthermore, the results are valid only for 8-node hypercubes. In order to assess the scalability of the methods, experiments on larger architectures are required. Because several of the communication strategies do not converge within 200 generations, a more sophisticated termination criteria based on convergence characteristics is recommended.

Appendix A. *MGA Generalization Experimental Data.*

This appendix contains the raw data from the generalized MGA performance experiments, including the fully deceptive binary function problem (Section A.1), the TSP (Section A.2), and Rosenbrock's Saddle (Section A.3). The data are summarized and interpreted in Chapter IV.

A.1 Fully Deceptive Function Data.

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	300	1	298	2	290	3	296
4	298	5	296	6	300	7	296
8	296	9	298	10	296	11	296
12	298	13	298	14	300	15	300
16	294	17	298	18	300	19	298
20	300	21	296	22	296	23	294
24	298	25	296	26	300	27	298
28	298	29	296	30	298	31	296
32	296	33	296	34	298	35	296
36	300	37	296	38	298	39	294

Table 41. Fully Deceptive Function Generalized Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	298	1	300	2	300	3	296
4	296	5	298	6	298	7	298
8	300	9	298	10	298	11	298
12	298	13	300	14	296	15	298
16	300	17	300	18	300	19	298
20	298	21	298	22	300	23	298
24	294	25	300	26	298	27	300
28	300	29	300	30	292	31	298
32	298	33	300	34	300	35	296
36	294	37	300	38	296	39	300

Table 42. Fully Deceptive Function Original Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	284	1	286	2	286	3	284
4	284	5	284	6	286	7	286
8	286	9	284	10	288	11	286
12	286	13	282	14	284	15	286
16	286	17	284	18	284	19	288
20	288	21	286	22	282	23	284
24	284	25	296	26	284	27	286
28	288	29	284	30	286	31	282
32	286	33	280	34	284	35	288
36	284	37	286	38	284	39	288

Table 43. Fully Deceptive Function Simple GA Best Perfomance

MEAN

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2030	0	0	0.509	1.498e+02	3.792e+01	3.080e+01	1.498e+02
1	3929	0	0	0.513	1.464e+02	3.298e+01	2.610e+01	1.424e+02
2	5834	0	0	0.519	1.438e+02	3.010e+01	2.330e+01	1.381e+02
3	7739	0	0	0.526	1.418e+02	2.817e+01	2.145e+01	1.352e+02
4	9637	0	0	0.533	1.402e+02	2.673e+01	2.030e+01	1.329e+02
5	11538	0	0	0.540	1.388e+02	2.551e+01	1.900e+01	1.311e+02
6	13435	0	0	0.549	1.375e+02	2.448e+01	1.755e+01	1.291e+02
7	15328	0	0	0.557	1.363e+02	2.361e+01	1.715e+01	1.272e+02
8	17220	0	0	0.566	1.351e+02	2.285e+01	1.620e+01	1.253e+02
9	19110	0	0	0.575	1.340e+02	2.218e+01	1.600e+01	1.231e+02
10	21005	0	0	0.585	1.329e+02	2.160e+01	1.545e+01	1.208e+02
11	22895	0	0	0.595	1.317e+02	2.109e+01	1.530e+01	1.186e+02
12	24786	0	0	0.605	1.306e+02	2.064e+01	1.525e+01	1.164e+02
13	26671	0	0	0.616	1.295e+02	2.026e+01	1.510e+01	1.140e+02
14	28559	0	0	0.626	1.283e+02	1.991e+01	1.490e+01	1.115e+02
15	30431	0	0	0.637	1.272e+02	1.960e+01	1.485e+01	1.090e+02
16	32306	0	0	0.648	1.260e+02	1.932e+01	1.475e+01	1.061e+02
17	34170	0	0	0.660	1.248e+02	1.907e+01	1.475e+01	1.032e+02
18	36031	0	0	0.671	1.236e+02	1.885e+01	1.470e+01	1.002e+02
19	37884	0	0	0.683	1.223e+02	1.865e+01	1.470e+01	9.720e+01
20	39734	0	0	0.694	1.210e+02	1.846e+01	1.465e+01	9.414e+01
21	41573	0	0	0.706	1.197e+02	1.829e+01	1.465e+01	9.070e+01

Table 44. Fully Deceptive Function Simple GA Performance Means by Generation

VARIANCE

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	0	0	0	0.000	9.958e-01	5.750e+01	4.755e+01	9.958e-01
0	253	0	0	0.000	9.152e-01	3.936e+01	4.307e+01	1.235e+00
0	495	0	0	0.000	8.703e-01	3.147e+01	3.334e+01	1.522e+00
0	593	0	0	0.000	7.096e-01	2.552e+01	2.195e+01	1.064e+00
0	705	0	0	0.000	6.571e-01	2.188e+01	1.550e+01	1.209e+00
0	900	0	0	0.000	6.226e-01	1.865e+01	1.169e+01	1.424e+00
0	1139	0	0	0.000	5.954e-01	1.597e+01	1.323e+01	1.356e+00
0	1482	0	0	0.000	5.418e-01	1.397e+01	1.228e+01	9.088e-01
0	1872	0	0	0.000	5.517e-01	1.248e+01	1.124e+01	1.516e+00
0	1638	0	0	0.000	5.918e-01	1.134e+01	1.046e+01	1.873e+00
0	1692	0	0	0.000	6.548e-01	1.037e+01	9.228e+00	2.385e+00
0	1780	0	0	0.000	7.338e-01	9.585e+00	9.138e+00	2.898e+00
0	1959	0	0	0.000	8.035e-01	9.017e+00	9.167e+00	3.099e+00
0	1945	0	0	0.000	8.635e-01	8.566e+00	8.400e+00	2.583e+00
0	2334	0	0	0.000	9.325e-01	8.170e+00	7.169e+00	3.725e+00
0	2404	0	0	0.000	1.014e+00	7.816e+00	6.746e+00	3.836e+00
0	2903	0	0	0.000	1.110e+00	7.493e+00	6.500e+00	4.943e+00
0	3448	0	0	0.000	1.245e+00	7.228e+00	6.500e+00	6.936e+00
0	4159	0	0	0.000	1.394e+00	6.989e+00	6.267e+00	7.022e+00
0	5370	0	0	0.000	1.517e+00	6.795e+00	6.267e+00	6.792e+00
0	5874	0	0	0.000	1.640e+00	6.633e+00	6.438e+00	6.352e+00
0	6180	0	0	0.000	1.777e+00	6.496e+00	6.438e+00	7.859e+00

Table 45. Fully Deceptive Function Simple GA Performance Variance by Generation

MAX

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2030	0	0	0.513	1.518e+02	5.094e+01	4.400e+01	1.518e+02
1	3958	0	0	0.518	1.484e+02	4.486e+01	4.200e+01	1.445e+02
2	5878	0	0	0.525	1.458e+02	4.279e+01	3.800e+01	1.401e+02
3	7782	0	0	0.533	1.437e+02	3.919e+01	3.200e+01	1.370e+02
4	9698	0	0	0.541	1.418e+02	3.658e+01	2.600e+01	1.352e+02
5	11598	0	0	0.549	1.404e+02	3.482e+01	2.600e+01	1.342e+02
6	13500	0	0	0.558	1.391e+02	3.290e+01	2.400e+01	1.322e+02
7	15424	0	0	0.568	1.379e+02	3.104e+01	2.400e+01	1.291e+02
8	17348	0	0	0.577	1.367e+02	2.961e+01	2.200e+01	1.281e+02
9	19202	0	0	0.587	1.356e+02	2.846e+01	2.200e+01	1.266e+02
10	21130	0	0	0.598	1.344e+02	2.753e+01	2.200e+01	1.245e+02
11	23016	0	0	0.608	1.334e+02	2.675e+01	2.200e+01	1.221e+02
12	24896	0	0	0.619	1.326e+02	2.607e+01	2.200e+01	1.220e+02
13	26768	0	0	0.630	1.317e+02	2.550e+01	2.200e+01	1.192e+02
14	28682	0	0	0.643	1.308e+02	2.500e+01	2.000e+01	1.166e+02
15	30550	0	0	0.654	1.297e+02	2.457e+01	2.000e+01	1.137e+02
16	32438	0	0	0.665	1.287e+02	2.412e+01	2.000e+01	1.118e+02
17	34326	0	0	0.677	1.277e+02	2.368e+01	2.000e+01	1.104e+02
18	36190	0	0	0.691	1.268e+02	2.328e+01	2.000e+01	1.084e+02
19	38070	0	0	0.701	1.256e+02	2.293e+01	2.000e+01	1.033e+02
20	39952	0	0	0.713	1.245e+02	2.270e+01	2.000e+01	9.982e+01
21	41790	0	0	0.725	1.233e+02	2.258e+01	2.000e+01	9.712e+01

Table 46. Fully Deceptive Function Simple GA Worst Performance by Generation

MIN Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2030	0	0	0.506	1.479e+02	1.438e+01	1.400e+01	1.479e+02
1	3898	0	0	0.510	1.442e+02	1.419e+01	1.400e+01	1.394e+02
2	5792	0	0	0.516	1.415e+02	1.413e+01	1.400e+01	1.357e+02
3	7694	0	0	0.521	1.400e+02	1.410e+01	1.400e+01	1.330e+02
4	9578	0	0	0.526	1.385e+02	1.408e+01	1.400e+01	1.308e+02
5	11460	0	0	0.531	1.372e+02	1.407e+01	1.400e+01	1.282e+02
6	13338	0	0	0.539	1.360e+02	1.406e+01	4.000e+00	1.267e+02
7	15214	0	0	0.547	1.348e+02	1.405e+01	4.000e+00	1.253e+02
8	17104	0	0	0.554	1.337e+02	1.404e+01	4.000e+00	1.233e+02
9	19014	0	0	0.562	1.325e+02	1.404e+01	4.000e+00	1.209e+02
10	20926	0	0	0.571	1.312e+02	1.404e+01	4.000e+00	1.179e+02
11	22824	0	0	0.579	1.301e+02	1.403e+01	4.000e+00	1.157e+02
12	24708	0	0	0.586	1.289e+02	1.403e+01	4.000e+00	1.135e+02
13	26582	0	0	0.597	1.276e+02	1.345e+01	4.000e+00	1.107e+02
14	28460	0	0	0.604	1.263e+02	1.283e+01	4.000e+00	1.068e+02
15	30350	0	0	0.612	1.249e+02	1.228e+01	4.000e+00	1.034e+02
16	32210	0	0	0.622	1.236e+02	1.180e+01	4.000e+00	1.015e+02
17	34070	0	0	0.633	1.223e+02	1.138e+01	4.000e+00	9.821e+01
18	35910	0	0	0.643	1.209e+02	1.100e+01	4.000e+00	9.458e+01
19	37736	0	0	0.656	1.195e+02	1.065e+01	4.000e+00	9.220e+01
20	39594	0	0	0.668	1.182e+02	1.034e+01	4.000e+00	8.883e+01
21	41450	0	0	0.679	1.168e+02	1.006e+01	4.000e+00	8.560e+01

Table 47. Fully Deceptive Function Simple GA Best Performance by Generation

A.2 Combinatoric Optimization Data.

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	1410	1	1503	2	1503	3	1410
4	1410	5	1410	6	1627	7	1410
8	1410	9	1410	10	1538	11	1699
12	1410	13	1410	14	1410	15	1410
16	1410	17	1410	18	1410	19	1410
20	1410	21	1410	22	1410	23	1410
24	1669	25	1410	26	1627	27	1503
28	1627	29	1410	30	1523	31	1410
32	1410	33	1410	34	1410	35	1523
36	1410	37	1590	38	1410	39	1410

Table 48. TSP Generalized Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	1000000	1	1000000	2	1000000	3	1000000
4	1000000	5	1000000	6	1000000	7	1000000
8	1000000	9	1000000	10	1000000	11	1000000
12	1000000	13	1000000	14	1000000	15	1000000
16	1000000	17	1000000	18	1000000	19	1000000
20	1000000	21	1000000	22	1000000	23	1000000
24	1000000	25	1000000	26	1000000	27	1000000
28	1000000	29	1000000	30	1000000	31	1000000
32	1000000	33	1000000	34	1000000	35	1000000
36	1000000	37	1000000	38	1000000	39	1000000

Table 49. TSP Original Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	1.0+06	1	1.0+06	2	1.0+06	3	1.0+06
4	1.0+06	5	1.0+06	6	1.0+06	7	1.0+06
8	1.0+06	9	1.0+06	10	1.0+06	11	1.0+06
12	1.0+06	13	1.0+06	14	1.0+06	15	1.0+06
16	1.0+06	17	1.0+06	18	1.0+06	19	1.0+06
20	1.0+06	21	1.0+06	22	1.0+06	23	1.0+06
24	1.0+06	25	1.0+06	26	1.0+06	27	1.0+06
28	1.0+06	29	1.0+06	30	1.0+06	31	1.0+06
32	1.0+06	33	1.0+06	34	1.0+06	35	1.0+06
36	1.0+06	37	1.0+06	38	1.0+06	39	1.0+06

Table 50. TSP Simple GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	2.025e+03	1	1.722e+03	2	2.209e+03	3	2.256e+03
4	1.912e+03	5	2.139e+03	6	1.777e+03	7	2.118e+03
8	2.234e+03	9	1.914e+03	10	1.777e+03	11	1.931e+03
12	1.914e+03	13	1.912e+03	14	2.025e+03	15	2.152e+03
16	1.972e+03	17	2.156e+03	18	1.914e+03	19	2.234e+03
20	1.777e+03	21	1.914e+03	22	2.330e+03	23	2.176e+03
24	2.048e+03	25	2.266e+03	26	1.867e+03	27	2.156e+03
28	1.972e+03	29	2.156e+03	30	2.221e+03	31	1.931e+03
32	2.235e+03	33	2.156e+03	34	2.340e+03	35	1.954e+03
36	1.969e+03	37	1.847e+03	38	1.889e+03	39	2.354e+03

Table 51. TSP Permutation GA Best Perfomance

MEAN Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
1	4392	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
2	6418	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
3	8440	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
4	10467	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
5	12485	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
6	14509	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
7	16527	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
8	18544	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
9	20564	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
10	22590	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
11	24610	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
12	26631	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
13	28650	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
14	30672	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
15	32694	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
16	34713	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
17	36730	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
18	38752	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
19	40769	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
20	42793	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
21	44818	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
22	46842	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
23	48859	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
24	50883	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
25	52905	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
26	54920	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
27	56944	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
28	58966	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
29	60983	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
30	63008	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
31	65030	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06
32	67056	0	0	0.508	1.000e+06	1.000e+06	1.000e+06	1.000e+06

Table 52. TSP Simple GA Performance Means by Generation

VARIANCE

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	0	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	552	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	1003	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	1759	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	1742	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	2474	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	3084	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	3595	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	4141	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	4051	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	4563	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	5376	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	5410	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	5914	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	6642	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	6950	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	7282	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	6289	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	6531	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	7362	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	7982	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	8012	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	8093	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	9598	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	9994	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	10540	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	10903	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	11632	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	12202	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	13792	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	12366	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	14153	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00
0	14665	0	0	0.000	0.000e+00	0.000e+00	0.000e+00	0.000e+00

Table 53. TSP Simple GA Performance Variance by Generation

MAX Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
1	4442	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
2	6478	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
3	8534	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
4	10572	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
5	12590	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
6	14620	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
7	16644	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
8	18660	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
9	20672	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
10	22708	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
11	24768	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
12	26790	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
13	28820	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
14	30876	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
15	32896	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
16	34940	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
17	36966	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
18	39010	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
19	41016	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
20	43060	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
21	45096	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
22	47112	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
23	49140	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
24	51130	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
25	53134	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
26	55148	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
27	57184	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
28	59208	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
29	61216	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
30	63218	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
31	65272	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06
32	67294	0	0	0.513	1.000e+06	1.000e+06	1.000e+06	1.000e+06

Table 54. TSP Simple GA Worst Performance by Generation

MIN

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
1	4338	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
2	6336	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
3	8352	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
4	10346	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
5	12362	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
6	14378	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
7	16396	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
8	18392	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
9	20390	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
10	22408	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
11	24422	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
12	26468	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
13	28446	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
14	30518	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
15	32506	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
16	34518	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
17	36554	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
18	38572	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
19	40586	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
20	42618	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
21	44620	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
22	46678	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
23	48684	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
24	50690	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
25	52680	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
26	54658	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
27	56696	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
28	58718	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
29	60706	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
30	62748	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
31	64780	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06
32	66834	0	0	0.505	1.000e+06	1.000e+06	1.000e+06	1.000e+06

Table 55. TSP Simple GA Best Performance by Generation

MEAN Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	1	0	0.965	5.793e+03	2.883e+03	2.607e+03	5.793e+03
1	4752	1	0	0.965	5.792e+03	2.684e+03	2.443e+03	5.792e+03
2	7128	1	0	0.965	5.791e+03	2.595e+03	2.380e+03	5.790e+03
3	9504	1	0	0.965	5.791e+03	2.538e+03	2.347e+03	5.791e+03
4	11880	1	0	0.965	5.791e+03	2.499e+03	2.328e+03	5.790e+03
5	14256	1	0	0.965	5.791e+03	2.466e+03	2.290e+03	5.791e+03
6	16632	1	0	0.965	5.791e+03	2.440e+03	2.281e+03	5.794e+03
7	19008	1	0	0.965	5.791e+03	2.420e+03	2.275e+03	5.792e+03
8	21384	1	0	0.965	5.791e+03	2.402e+03	2.257e+03	5.790e+03
9	23760	1	0	0.965	5.791e+03	2.387e+03	2.222e+03	5.793e+03
10	26136	1	0	0.965	5.792e+03	2.370e+03	2.193e+03	5.793e+03
11	28512	1	0	0.965	5.791e+03	2.354e+03	2.182e+03	5.789e+03
12	30888	1	0	0.965	5.791e+03	2.340e+03	2.161e+03	5.788e+03
13	33264	1	0	0.965	5.791e+03	2.327e+03	2.152e+03	5.786e+03
14	35640	1	0	0.965	5.791e+03	2.315e+03	2.152e+03	5.790e+03
15	38016	1	0	0.965	5.791e+03	2.305e+03	2.148e+03	5.790e+03
16	40392	1	0	0.965	5.791e+03	2.295e+03	2.141e+03	5.792e+03
17	42768	1	0	0.965	5.790e+03	2.286e+03	2.120e+03	5.785e+03
18	45144	1	0	0.965	5.790e+03	2.277e+03	2.118e+03	5.790e+03
19	47520	1	0	0.965	5.790e+03	2.269e+03	2.113e+03	5.790e+03
20	49896	1	0	0.965	5.790e+03	2.261e+03	2.108e+03	5.793e+03
21	52272	1	0	0.965	5.790e+03	2.254e+03	2.097e+03	5.790e+03
22	54648	1	0	0.965	5.790e+03	2.247e+03	2.092e+03	5.788e+03
23	57024	1	0	0.965	5.790e+03	2.240e+03	2.068e+03	5.790e+03
24	59400	1	0	0.965	5.790e+03	2.233e+03	2.068e+03	5.793e+03
25	61776	1	0	0.965	5.790e+03	2.227e+03	2.059e+03	5.787e+03
26	64152	1	0	0.965	5.790e+03	2.220e+03	2.059e+03	5.789e+03
27	66528	1	0	0.965	5.790e+03	2.215e+03	2.046e+03	5.785e+03

Table 56. TSP Permutation GA Performance Means by Generation

VARIANCE

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	0	0	0	0.000	4.291e+02	3.446e+04	8.161e+04	4.291e+02
0	0	0	0	0.000	1.708e+02	3.050e+04	4.379e+04	3.683e+02
0	0	0	0	0.000	1.064e+02	2.822e+04	3.258e+04	4.680e+02
0	0	0	0	0.000	8.713e+01	2.524e+04	2.504e+04	4.975e+02
0	0	0	0	0.000	5.813e+01	2.252e+04	3.255e+04	3.920e+02
0	0	0	0	0.000	5.940e+01	2.121e+04	3.714e+04	5.818e+02
0	0	0	0	0.000	5.538e+01	2.076e+04	3.585e+04	3.492e+02
0	0	0	0	0.000	5.021e+01	2.088e+04	3.520e+04	5.310e+02
0	0	0	0	0.000	4.832e+01	2.094e+04	3.591e+04	3.466e+02
0	0	0	0	0.000	4.461e+01	2.100e+04	3.564e+04	3.195e+02
0	0	0	0	0.000	3.890e+01	2.015e+04	2.907e+04	3.284e+02
0	0	0	0	0.000	3.637e+01	1.940e+04	2.729e+04	3.790e+02
0	0	0	0	0.000	3.206e+01	1.845e+04	2.914e+04	3.865e+02
0	0	0	0	0.000	2.653e+01	1.763e+04	3.151e+04	4.946e+02
0	0	0	0	0.000	2.612e+01	1.709e+04	3.151e+04	4.287e+02
0	0	0	0	0.000	2.867e+01	1.679e+04	3.068e+04	3.850e+02
0	0	0	0	0.000	2.724e+01	1.672e+04	3.168e+04	5.011e+02
0	0	0	0	0.000	2.772e+01	1.670e+04	3.509e+04	3.994e+02
0	0	0	0	-0.000	2.600e+01	1.675e+04	3.415e+04	3.937e+02
0	0	0	0	-0.000	2.674e+01	1.678e+04	3.266e+04	4.078e+02
0	0	0	0	0.000	2.430e+01	1.688e+04	3.169e+04	3.453e+02
0	0	0	0	0.000	2.284e+01	1.694e+04	3.058e+04	4.518e+02
0	0	0	0	0.000	2.387e+01	1.699e+04	2.908e+04	3.421e+02
0	0	0	0	0.000	2.097e+01	1.696e+04	3.110e+04	3.787e+02
0	0	0	0	0.000	1.925e+01	1.701e+04	3.110e+04	3.756e+02
0	0	0	0	0.000	1.862e+01	1.705e+04	3.043e+04	3.789e+02
0	0	0	0	0.000	1.603e+01	1.710e+04	3.043e+04	3.289e+02
0	0	0	0	0.000	1.691e+01	1.713e+04	3.039e+04	5.561e+02

Table 57. TSP Permutation GA Performance Variance by Generation

MAX Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	1	0	0.965	5.829e+03	3.232e+03	3.152e+03	5.829e+03
1	4752	1	0	0.965	5.829e+03	3.080e+03	2.975e+03	5.835e+03
2	7128	1	0	0.965	5.818e+03	2.964e+03	2.808e+03	5.831e+03
3	9504	1	0	0.965	5.812e+03	2.856e+03	2.673e+03	5.837e+03
4	11880	1	0	0.965	5.807e+03	2.791e+03	2.673e+03	5.836e+03
5	14256	1	0	0.965	5.814e+03	2.751e+03	2.588e+03	5.847e+03
6	16632	1	0	0.965	5.816e+03	2.728e+03	2.588e+03	5.827e+03
7	19008	1	0	0.965	5.813e+03	2.710e+03	2.588e+03	5.850e+03
8	21384	1	0	0.965	5.812e+03	2.697e+03	2.588e+03	5.819e+03
9	23760	1	0	0.965	5.811e+03	2.686e+03	2.588e+03	5.844e+03
10	26136	1	0	0.965	5.811e+03	2.668e+03	2.492e+03	5.825e+03
11	28512	1	0	0.965	5.808e+03	2.641e+03	2.469e+03	5.828e+03
12	30888	1	0	0.965	5.807e+03	2.592e+03	2.469e+03	5.828e+03
13	33264	1	0	0.965	5.804e+03	2.544e+03	2.469e+03	5.828e+03
14	35640	1	0	0.965	5.803e+03	2.531e+03	2.469e+03	5.831e+03
15	38016	1	0	0.965	5.803e+03	2.519e+03	2.469e+03	5.826e+03
16	40392	1	0	0.965	5.802e+03	2.509e+03	2.469e+03	5.840e+03
17	42768	1	0	0.965	5.801e+03	2.506e+03	2.469e+03	5.847e+03
18	45144	1	0	0.965	5.801e+03	2.503e+03	2.421e+03	5.827e+03
19	47520	1	0	0.965	5.802e+03	2.499e+03	2.421e+03	5.832e+03
20	49896	1	0	0.965	5.801e+03	2.495e+03	2.421e+03	5.847e+03
21	52272	1	0	0.965	5.801e+03	2.489e+03	2.354e+03	5.835e+03
22	54648	1	0	0.965	5.802e+03	2.483e+03	2.354e+03	5.828e+03
23	57024	1	0	0.965	5.800e+03	2.478e+03	2.354e+03	5.844e+03
24	59400	1	0	0.965	5.798e+03	2.473e+03	2.354e+03	5.827e+03
25	61776	1	0	0.965	5.799e+03	2.468e+03	2.354e+03	5.826e+03
26	64152	1	0	0.965	5.798e+03	2.464e+03	2.354e+03	5.827e+03
27	66528	1	0	0.965	5.798e+03	2.460e+03	2.354e+03	5.829e+03

Table 58. TSP Permutation GA Worst Performance by Generation

MIN Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2376	1	0	0.964	5.752e+03	2.506e+03	2.086e+03	5.752e+03
1	4752	1	0	0.964	5.769e+03	2.358e+03	2.086e+03	5.750e+03
2	7128	1	0	0.964	5.769e+03	2.278e+03	2.086e+03	5.738e+03
3	9504	1	0	0.964	5.776e+03	2.238e+03	2.086e+03	5.747e+03
4	11880	1	0	0.964	5.772e+03	2.214e+03	1.722e+03	5.751e+03
5	14256	1	0	0.964	5.778e+03	2.198e+03	1.722e+03	5.732e+03
6	16632	1	0	0.964	5.778e+03	2.184e+03	1.722e+03	5.747e+03
7	19008	1	0	0.964	5.779e+03	2.127e+03	1.722e+03	5.746e+03
8	21384	1	0	0.964	5.779e+03	2.082e+03	1.722e+03	5.743e+03
9	23760	1	0	0.964	5.779e+03	2.045e+03	1.722e+03	5.752e+03
10	26136	1	0	0.964	5.782e+03	2.016e+03	1.722e+03	5.748e+03
11	28512	1	0	0.964	5.780e+03	1.991e+03	1.722e+03	5.745e+03
12	30888	1	0	0.964	5.782e+03	1.970e+03	1.722e+03	5.739e+03
13	33264	1	0	0.964	5.782e+03	1.952e+03	1.722e+03	5.741e+03
14	35640	1	0	0.964	5.780e+03	1.937e+03	1.722e+03	5.747e+03
15	38016	1	0	0.964	5.779e+03	1.923e+03	1.722e+03	5.742e+03
16	40392	1	0	0.964	5.781e+03	1.911e+03	1.722e+03	5.749e+03
17	42768	1	0	0.964	5.780e+03	1.901e+03	1.722e+03	5.752e+03
18	45144	1	0	0.965	5.781e+03	1.891e+03	1.722e+03	5.746e+03
19	47520	1	0	0.965	5.781e+03	1.883e+03	1.722e+03	5.743e+03
20	49896	1	0	0.964	5.780e+03	1.875e+03	1.722e+03	5.761e+03
21	52272	1	0	0.964	5.781e+03	1.868e+03	1.722e+03	5.747e+03
22	54648	1	0	0.964	5.780e+03	1.862e+03	1.722e+03	5.761e+03
23	57024	1	0	0.964	5.782e+03	1.856e+03	1.722e+03	5.745e+03
24	59400	1	0	0.964	5.782e+03	1.850e+03	1.722e+03	5.742e+03
25	61776	1	0	0.964	5.781e+03	1.845e+03	1.722e+03	5.741e+03
26	64152	1	0	0.964	5.782e+03	1.841e+03	1.722e+03	5.738e+03
27	66528	1	0	0.964	5.782e+03	1.836e+03	1.722e+03	5.722e+03

Table 59. TSP Permutation GA Best Performance by Generation

A.3 Functional Optimization Data.

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	0.026092	1	0.020743	2	0.000639	3	0.011878
4	0.000049	5	0.012921	6	0.008695	7	0.006776
8	0.015415	9	0.009012	10	0.006176	11	0.015256
12	0.009221	13	0.002241	14	0.003616	15	0.003507
16	0.010536	17	0.027167	18	0.005836	19	0.009364
20	0.009586	21	0.005694	22	0.007566	23	0.002265
24	0.007591	25	0.004638	26	0.006271	27	0.009316
28	0.007751	29	0.000551	30	0.008238	31	0.023999
32	0.005836	33	0.004717	34	0.003435	35	0.000146
36	0.002754	37	0.006592	38	0.002874	39	0.008700

Table 60. Rosenbrock's Saddle Generalized Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
1	0.344996	2	0.012690	3	0.000113	4	0.002025
5	0.078416	6	0.076939	7	0.000001	8	0.091986
9	0.088213	10	0.065989	11	0.065989	12	0.000594
13	0.085796	14	0.063520	15	1.002101	16	0.065989
17	0.223559	18	0.000594	19	0.657135	20	0.065989
21	1.002101	22	0.239122	23	0.088742	24	0.000594
25	1.002101	26	0.024123	27	0.001964	28	0.057616
29	0.093537	30	1.518744	31	0.000146	32	0.239198
33	0.129299	34	1.002101	35	0.007653	36	0.342571
37	1.002101	38	0.109685	39	1.037597	40	1.332488

Table 61. Rosenbrock's Saddle Original Messy GA Best Perfomance

Trial	Best solution	Trial	Best solution	Trial	Best solution	Trial	Best solution
0	3.344976e-04	1	5.960000e-04	2	8.165610e-05	3	1.289296e-04
4	4.350625e-04	5	1.300625e-04	6	4.160000e-04	7	1.602560e-05
8	4.014001e-04	9	1.654561e-04	10	5.817521e-04	11	1.602560e-05
12	1.718561e-04	13	1.772096e-04	14	1.128256e-04	15	1.012001e-04
16	2.748736e-04	17	5.960000e-04	18	4.296401e-04	19	9.577296e-04
20	4.018321e-04	21	9.611521e-04	22	1.072081e-04	23	2.606416e-04
24	4.388096e-04	25	4.217296e-04	26	1.072081e-04	27	3.740321e-04
28	3.390416e-04	29	7.216976e-04	30	1.516096e-04	31	4.296401e-04
32	1.998416e-04	33	4.014001e-04	34	2.172736e-04	35	2.172736e-04
36	1.010000e-04	37	1.300625e-04	38	6.864976e-04	39	1.772096e-04

Table 62. Rosenbrock's Saddle Simple GA Best Perfomance

MEAN Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2024	0	0	0.509	4.959e+02	5.229e-01	1.819e-02	4.959e+02
2	5725	0	0	0.514	4.071e+02	1.929e-01	6.930e-03	3.273e+02
3	7579	0	0	0.517	3.799e+02	1.471e-01	4.223e-03	2.926e+02
4	9430	0	0	0.519	3.585e+02	1.190e-01	3.887e-03	2.673e+02
5	11277	0	0	0.520	3.412e+02	1.001e-01	3.546e-03	2.488e+02
6	13133	0	0	0.522	3.264e+02	8.640e-02	2.587e-03	2.341e+02
7	14984	0	0	0.523	3.140e+02	7.603e-02	2.466e-03	2.232e+02
8	16834	0	0	0.524	3.030e+02	6.794e-02	2.216e-03	2.119e+02
9	18679	0	0	0.525	2.938e+02	6.145e-02	1.976e-03	2.056e+02
10	20527	0	0	0.526	2.854e+02	5.608e-02	1.797e-03	1.983e+02
11	22374	0	0	0.527	2.781e+02	5.159e-02	1.523e-03	1.938e+02
12	24221	0	0	0.528	2.714e+02	4.776e-02	1.408e-03	1.880e+02
13	26304	0	0	0.529	2.648e+02	4.410e-02	1.148e-03	1.834e+02
15	29540	0	0	0.531	2.555e+02	3.939e-02	9.150e-04	1.768e+02
16	31617	0	0	0.532	2.503e+02	3.685e-02	8.193e-04	1.733e+02
17	33466	0	0	0.532	2.460e+02	3.486e-02	7.636e-04	1.705e+02
18	35310	0	0	0.533	2.420e+02	3.308e-02	7.636e-04	1.662e+02
19	37155	0	0	0.533	2.382e+02	3.147e-02	7.320e-04	1.634e+02
20	38998	0	0	0.533	2.347e+02	3.002e-02	6.806e-04	1.621e+02
21	40844	0	0	0.534	2.314e+02	2.869e-02	5.780e-04	1.591e+02
22	42686	0	0	0.535	2.282e+02	2.747e-02	5.527e-04	1.568e+02
23	44527	0	0	0.535	2.253e+02	2.636e-02	5.352e-04	1.549e+02
24	46372	0	0	0.536	2.225e+02	2.533e-02	5.299e-04	1.544e+02
25	48212	0	0	0.536	2.199e+02	2.439e-02	5.294e-04	1.520e+02
26	50566	0	0	0.536	2.168e+02	2.327e-02	5.255e-04	1.503e+02
28	53418	0	0	0.537	2.132e+02	2.207e-02	4.774e-04	1.477e+02
29	55582	0	0	0.537	2.106e+02	2.122e-02	4.656e-04	1.457e+02
30	57423	0	0	0.537	2.086e+02	2.055e-02	4.459e-04	1.449e+02
31	59262	0	0	0.538	2.066e+02	1.993e-02	4.236e-04	1.428e+02
32	61103	0	0	0.538	2.047e+02	1.934e-02	4.182e-04	1.417e+02
33	62942	0	0	0.538	2.029e+02	1.879e-02	4.110e-04	1.394e+02
34	64784	0	0	0.539	2.011e+02	1.826e-02	3.693e-04	1.398e+02
35	66623	0	1	0.539	1.994e+02	1.777e-02	3.668e-04	1.385e+02
36	68465	0	1	0.539	1.978e+02	1.730e-02	3.313e-04	1.381e+02
37	70304	0	1	0.539	1.963e+02	1.686e-02	3.242e-04	1.376e+02

Table 63. Rosenbrock's Saddle Simple GA Performance Means by Generation

VARIANCE

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	0	0	0	0.000	1.472e+02	1.274e-01	2.774e-04	1.472e+02
0	704	0	0	0.000	9.614e+01	1.573e-02	3.170e-05	1.549e+02
0	1109	0	0	0.000	8.495e+01	9.005e-03	1.858e-05	1.354e+02
0	1431	0	0	0.000	7.168e+01	5.855e-03	1.673e-05	8.680e+01
0	1653	0	0	0.000	6.223e+01	4.113e-03	1.347e-05	8.162e+01
0	2174	0	0	0.000	5.792e+01	3.046e-03	6.686e-06	1.361e+02
0	2332	0	0	0.000	5.270e+01	2.341e-03	6.548e-06	9.796e+01
0	2821	0	0	0.000	4.976e+01	1.856e-03	5.768e-06	7.530e+01
0	3393	0	0	0.000	4.756e+01	1.510e-03	4.651e-06	7.454e+01
0	4032	0	0	0.000	4.568e+01	1.251e-03	4.550e-06	6.651e+01
0	4605	0	0	0.000	4.474e+01	1.055e-03	2.897e-06	7.903e+01
0	4272	0	0	0.000	4.277e+01	8.998e-04	2.942e-06	8.804e+01
0	339046	0	0	0.000	4.410e+01	7.674e-04	1.256e-06	9.000e+01
0	335845	0	0	0.000	3.854e+01	5.985e-04	4.913e-07	6.792e+01
0	3815	0	0	0.000	3.410e+01	5.292e-04	3.819e-07	4.093e+01
0	3436	0	0	0.000	3.235e+01	4.728e-04	3.748e-07	4.451e+01
0	3959	0	0	0.000	3.062e+01	4.250e-04	3.748e-07	4.396e+01
0	5058	0	0	0.000	2.870e+01	3.842e-04	3.748e-07	4.741e+01
0	5766	0	0	0.000	2.793e+01	3.487e-04	3.677e-07	6.017e+01
0	5749	0	0	0.000	2.697e+01	3.180e-04	1.958e-07	4.789e+01
0	5691	0	0	0.000	2.658e+01	2.911e-04	1.876e-07	4.750e+01
0	5965	0	0	0.000	2.617e+01	2.674e-04	1.805e-07	4.326e+01
0	5860	0	0	0.000	2.564e+01	2.467e-04	1.806e-07	5.656e+01
0	6370	0	0	0.000	2.466e+01	2.282e-04	1.807e-07	4.540e+01
0	603041	0	0	0.000	2.345e+01	2.061e-04	1.802e-07	5.565e+01
0	426513	0	0	0.000	2.450e+01	1.862e-04	8.773e-08	4.798e+01
0	8600	0	0	0.000	2.219e+01	1.718e-04	8.279e-08	5.404e+01
0	9093	0	0	0.000	2.168e+01	1.609e-04	8.680e-08	3.267e+01
0	9287	0	0	0.000	2.104e+01	1.511e-04	8.122e-08	3.739e+01
0	9862	0	0	0.000	2.041e+01	1.421e-04	7.687e-08	3.793e+01
0	9830	0	0	0.000	1.978e+01	1.340e-04	7.920e-08	4.763e+01
0	11078	0	0	0.000	1.937e+01	1.265e-04	6.143e-08	6.605e+01
0	11943	0	0	0.000	1.901e+01	1.196e-04	6.165e-08	5.700e+01
0	12111	0	0	0.000	1.844e+01	1.133e-04	5.467e-08	4.066e+01
0	12173	0	0	0.000	1.791e+01	1.074e-04	5.548e-08	2.462e+01

Table 64. Rosenbrock's Saddle Simple GA Performance Variance by Generation

MAX Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2024	0	0	0.512	5.265e+02	1.436e+00	6.678e-02	5.265e+02
2	5772	0	0	0.518	4.311e+02	5.171e-01	2.234e-02	3.525e+02
3	7650	0	0	0.521	4.005e+02	3.926e-01	1.711e-02	3.120e+02
4	9522	0	0	0.523	3.751e+02	3.188e-01	1.711e-02	2.877e+02
5	11364	0	0	0.525	3.584e+02	2.681e-01	1.385e-02	2.716e+02
6	13234	0	0	0.527	3.460e+02	2.313e-01	1.018e-02	2.653e+02
7	15104	0	0	0.528	3.335e+02	2.032e-01	1.018e-02	2.500e+02
8	16956	0	0	0.530	3.224e+02	1.811e-01	1.018e-02	2.296e+02
9	18818	0	0	0.531	3.119e+02	1.634e-01	1.018e-02	2.211e+02
10	20666	0	0	0.533	3.030e+02	1.489e-01	1.018e-02	2.179e+02
11	22528	0	0	0.534	2.947e+02	1.368e-01	9.414e-03	2.138e+02
12	24352	0	0	0.535	2.873e+02	1.265e-01	9.414e-03	2.091e+02
14	27846	0	0	0.536	2.807e+02	1.176e-01	5.661e-03	2.013e+02
15	29840	0	0	0.538	2.684e+02	1.029e-01	3.377e-03	1.934e+02
16	31754	0	0	0.538	2.626e+02	9.697e-02	3.377e-03	1.849e+02
17	33578	0	0	0.539	2.574e+02	9.173e-02	3.377e-03	1.830e+02
18	35438	0	0	0.539	2.526e+02	8.698e-02	3.377e-03	1.831e+02
19	37286	0	0	0.539	2.488e+02	8.270e-02	3.377e-03	1.824e+02
20	39126	0	0	0.540	2.455e+02	7.876e-02	3.377e-03	1.783e+02
21	40976	0	0	0.541	2.422e+02	7.526e-02	2.362e-03	1.726e+02
22	42816	0	0	0.542	2.392e+02	7.201e-02	2.362e-03	1.710e+02
23	44656	0	0	0.542	2.360e+02	6.904e-02	2.362e-03	1.689e+02
24	46514	0	0	0.542	2.335e+02	6.632e-02	2.362e-03	1.685e+02
25	48360	0	0	0.543	2.308e+02	6.380e-02	2.362e-03	1.667e+02
27	51866	0	0	0.544	2.284e+02	5.933e-02	2.362e-03	1.641e+02
28	53826	0	0	0.544	2.228e+02	5.734e-02	1.230e-03	1.620e+02
29	55804	0	0	0.544	2.201e+02	5.544e-02	1.230e-03	1.621e+02
30	57656	0	0	0.544	2.179e+02	5.367e-02	1.230e-03	1.609e+02
31	59506	0	1	0.545	2.156e+02	5.201e-02	1.230e-03	1.606e+02
32	61348	0	1	0.546	2.133e+02	5.043e-02	1.230e-03	1.519e+02
33	63156	0	1	0.546	2.111e+02	4.895e-02	1.230e-03	1.517e+02
34	65012	0	1	0.546	2.093e+02	4.758e-02	1.003e-03	1.531e+02
35	66864	0	1	0.545	2.075e+02	4.627e-02	1.003e-03	1.534e+02
36	68728	0	1	0.546	2.058e+02	4.505e-02	9.612e-04	1.526e+02
37	70576	0	1	0.546	2.041e+02	4.386e-02	9.612e-04	1.474e+02

Table 65. Rosenbrock's Saddle Simple GA Worst Performance by Generation

MIN

Gens	Trials	Lost	Conv	Entr	Online	Offline	Best	Average
0	2024	0	0	0.506	4.644e+02	4.741e-02	7.217e-04	4.644e+02
2	5658	0	0	0.510	3.808e+02	2.755e-02	1.461e-04	2.946e+02
3	7502	0	0	0.511	3.580e+02	2.206e-02	1.010e-04	2.688e+02
4	9346	0	0	0.514	3.375e+02	1.845e-02	1.010e-04	2.471e+02
5	11182	0	0	0.515	3.221e+02	1.610e-02	1.010e-04	2.287e+02
6	13018	0	0	0.517	3.079e+02	1.425e-02	1.010e-04	2.112e+02
7	14884	0	0	0.518	2.958e+02	1.267e-02	1.010e-04	2.102e+02
8	16726	0	0	0.520	2.853e+02	1.143e-02	1.010e-04	1.959e+02
9	18554	0	0	0.520	2.757e+02	1.046e-02	1.010e-04	1.848e+02
10	20380	0	0	0.521	2.683e+02	9.660e-03	1.010e-04	1.835e+02
11	22208	0	0	0.522	2.612e+02	8.999e-03	1.010e-04	1.784e+02
12	24066	0	0	0.523	2.544e+02	8.428e-03	1.010e-04	1.693e+02
13	26006	0	0	0.524	2.480e+02	7.918e-03	1.010e-04	1.618e+02
14	28002	0	0	0.526	2.385e+02	6.950e-03	1.010e-04	1.563e+02
16	31496	0	0	0.527	2.342e+02	6.556e-03	8.166e-05	1.592e+02
17	33348	0	0	0.528	2.307e+02	6.199e-03	8.166e-05	1.594e+02
18	35178	0	0	0.529	2.273e+02	5.885e-03	8.166e-05	1.548e+02
19	37008	0	0	0.529	2.238e+02	5.599e-03	8.166e-05	1.529e+02
20	38806	0	0	0.529	2.204e+02	5.343e-03	8.166e-05	1.461e+02
21	40678	0	0	0.530	2.172e+02	5.108e-03	8.166e-05	1.445e+02
22	42502	0	0	0.530	2.145e+02	4.896e-03	8.166e-05	1.457e+02
23	44360	0	0	0.530	2.121e+02	4.702e-03	8.166e-05	1.430e+02
24	46214	0	0	0.531	2.097e+02	4.519e-03	1.603e-05	1.345e+02
25	48050	0	0	0.532	2.075e+02	4.356e-03	1.603e-05	1.375e+02
26	50004	0	0	0.532	2.054e+02	4.203e-03	1.603e-05	1.344e+02
27	52000	0	0	0.532	2.012e+02	4.060e-03	1.603e-05	1.291e+02
29	55370	0	0	0.532	1.991e+02	3.805e-03	1.603e-05	1.287e+02
30	57218	0	0	0.533	1.972e+02	3.690e-03	1.603e-05	1.344e+02
31	59062	0	0	0.533	1.953e+02	3.579e-03	1.603e-05	1.336e+02
32	60912	0	0	0.533	1.934e+02	3.475e-03	1.603e-05	1.290e+02
33	62758	0	0	0.533	1.916e+02	3.380e-03	1.603e-05	1.231e+02
34	64586	0	0	0.533	1.897e+02	3.289e-03	1.603e-05	1.238e+02
35	66420	0	0	0.534	1.880e+02	3.203e-03	1.603e-05	1.244e+02
36	68266	0	0	0.534	1.867e+02	3.122e-03	1.603e-05	1.240e+02
37	70118	0	0	0.534	1.852e+02	3.046e-03	1.603e-05	1.259e+02

Table 66. Rosenbrock's Saddle Simple GA Best Performance by Generation

Appendix B. *MGA Parallel Decomposition Experimental Data.*

This appendix contains the raw data from the parallel MGA data distribution performance experiments, including the indexed distribution strategy (Section B.1), the modified indexed distribution strategy (Section B.2), the interleaved distribution strategy (Section B.3), and the block distribution strategy (Section B.4). Each table gives the average over all applicable experiments of the maximum time spent by any processor in each operation. Note that the "Convert Data Structure" operation includes the time required to recombine the subpopulations prior to the juxtapositional phase. The "Distribution Dependent Time" given is the average over all applicable experiments of the maximum time spent by any processor in the initialization of the population, the primordial phase, and the data structure conversion. Likewise, the "Total" time is the average over all applicable experiments of the maximum total execution time of any processor. The data are summarized and interpreted in Chapter V.

B.1 Indexed Distribution.

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.10e-02	1.83e-18
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	1.64e+01	5.33e-02
Primordial Phase	1.55e+02	3.89e-01
Convert Data Structure	6.18e-01	7.38e-04
Distribution Dependent Time	1.72e+02	3.88e-01
Juxtapositional Phase	7.69e+01	6.73e+00
Other	6.20e-03	2.15e-03
Total	2.49e+02	6.72e+00

Table 67. Indexed Distribution 1-Node Execution Times

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.11e-02	3.16e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	8.69e+00	2.89e-02
Primordial Phase	7.07e+01	2.80e-01
Convert Data Structure	1.08e+01	2.09e-01
Distribution Dependent Time	9.02e+01	4.16e-01
Juxtapositional Phase	8.22e+01	8.60e+00
Other	2.20e-03	4.22e-04
Total	1.63e+02	8.56e+00

Table 68. Indexed Distribution 2-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.92e-01	2.64e-02
Create Building Blocks	1.00e+00	0.00e+00
Initialize Population	1.89e+00	3.01e-04
Primordial Phase	2.19e+00	7.32e-03
Convert Data Structure	5.71e-02	1.13e-03
Distribution Dependent Time	1.91e+00	8.88e-03
Juxtapositional Phase	9.41e-01	9.89e-02
Other	2.95e+00	1.24e+00
Total	1.53e+00	7.63e-02

Table 69. Indexed Distribution 2-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.11e-02	3.16e-04
Create Building Blocks	1.10e-03	3.16e-04
Initialize Population	4.85e+00	1.56e-02
Primordial Phase	3.10e+01	1.80e-01
Convert Data Structure	1.24e+01	1.72e-01
Distribution Dependent Time	4.82e+01	3.30e-01
Juxtapositional Phase	8.57e+01	5.88e+00
Other	2.40e-03	5.16e-04
Total	1.23e+02	5.84e+00

Table 70. Indexed Distribution 4-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.92e-01	2.64e-02
Create Building Blocks	9.50e-01	1.58e-01
Initialize Population	3.38e+00	6.14e-04
Primordial Phase	5.00e+00	2.52e-02
Convert Data Structure	4.98e-02	7.16e-04
Distribution Dependent Time	3.56e+00	2.07e-02
Juxtapositional Phase	8.99e-01	7.93e-02
Other	2.73e+00	1.26e+00
Total	2.03e+00	9.36e-02

Table 71. Indexed Distribution 4-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.10e-03	3.16e-04
Initialize Population	2.97e+00	9.23e-03
Primordial Phase	1.38e+01	1.69e-01
Convert Data Structure	1.07e+01	1.76e-01
Distribution Dependent Time	2.74e+01	3.26e-01
Juxtapositional Phase	9.43e+01	1.37e+01
Other	2.70e-03	4.83e-04
Total	1.12e+02	1.36e+01

Table 72. Indexed Distribution 8-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.83e-01	3.51e-02
Create Building Blocks	9.50e-01	1.58e-01
Initialize Population	5.52e+00	1.22e-03
Primordial Phase	1.13e+01	1.27e-01
Convert Data Structure	5.77e-02	9.63e-04
Distribution Dependent Time	6.27e+00	6.87e-02
Juxtapositional Phase	8.28e-01	1.20e-01
Other	2.32e+00	6.96e-01
Total	2.24e+00	2.40e-01

Table 73. Indexed Distribution 8-Node Speedups

B.2 Modified Indexed Distribution.

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.07e-02	6.75e-04
Create Building Blocks	1.30e-03	4.83e-04
Initialize Population	1.64e+01	5.45e-02
Primordial Phase	1.55e+02	3.78e-01
Convert Data Structure	6.18e-01	5.68e-04
Distribution Dependent Time	1.72e+02	3.75e-01
Juxtapositional Phase	7.69e+01	6.73e+00
Other	5.10e-03	1.37e-03
Total	2.49e+02	6.71e+00

Table 74. Modified Indexed Distribution 1-Node Execution Times

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	8.58e+00	2.84e-02
Primordial Phase	6.92e+01	3.92e-01
Convert Data Structure	8.07e+00	4.15e-01
Distribution Dependent Time	8.58e+01	7.47e-01
Juxtapositional Phase	7.97e+01	5.24e+00
Other	2.10e-03	3.16e-04
Total	1.58e+02	5.24e+00

Table 75. Modified Indexed Distribution 2-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.55e-01	4.72e-02
Create Building Blocks	1.30e+00	4.83e-01
Initialize Population	1.91e+00	3.65e-04
Primordial Phase	2.24e+00	1.15e-02
Convert Data Structure	7.67e-02	4.05e-03
Distribution Dependent Time	2.00e+00	1.66e-02
Juxtapositional Phase	9.64e-01	4.68e-02
Other	2.45e+00	6.85e-01
Total	1.57e+00	3.04e-02

Table 76. Modified Indexed Distribution 2-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.11e-02	3.16e-04
Create Building Blocks	1.20e-03	4.22e-04
Initialize Population	4.53e+00	1.49e-02
Primordial Phase	2.87e+01	2.90e-01
Convert Data Structure	7.70e+00	3.60e-01
Distribution Dependent Time	4.09e+01	6.25e-01
Juxtapositional Phase	8.35e+01	6.65e+00
Other	2.40e-03	5.16e-04
Total	1.18e+02	6.76e+00

Table 77. Modified Indexed Distribution 4-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.64e-01	4.69e-02
Create Building Blocks	1.15e+00	4.74e-01
Initialize Population	3.62e+00	1.01e-03
Primordial Phase	5.40e+00	5.27e-02
Convert Data Structure	8.04e-02	3.89e-03
Distribution Dependent Time	4.20e+00	6.34e-02
Juxtapositional Phase	9.25e-01	9.52e-02
Other	2.15e+00	5.52e-01
Total	2.12e+00	1.22e-01

Table 78. Modified Indexed Distribution 4-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.10e-03	3.16e-04
Initialize Population	2.26e+00	7.07e-03
Primordial Phase	1.08e+01	1.61e-01
Convert Data Structure	4.10e+00	2.02e-01
Distribution Dependent Time	1.71e+01	3.52e-01
Juxtapositional Phase	9.67e+01	1.34e+01
Other	2.50e-03	5.27e-04
Total	1.11e+02	1.34e+01

Table 79. Modified Indexed Distribution 8-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.55e-01	4.72e-02
Create Building Blocks	1.25e+00	5.40e-01
Initialize Population	7.26e+00	2.32e-03
Primordial Phase	1.44e+01	2.19e-01
Convert Data Structure	1.51e-01	7.30e-03
Distribution Dependent Time	1.00e+01	2.08e-01
Juxtapositional Phase	8.04e-01	9.50e-02
Other	2.08e+00	6.59e-01
Total	2.27e+00	2.43e-01

Table 80. Modified Indexed Distribution 8-Node Speedups

B.3 Interleaved Distribution.

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.08e-02	4.22e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	1.65e+01	6.15e-02
Primordial Phase	1.55e+02	5.96e-01
Convert Data Structure	6.20e-01	6.32e-04
Distribution Dependent Time	1.72e+02	6.19e-01
Juxtapositional Phase	8.03e+01	1.58e+01
Other	5.30e-03	1.57e-03
Total	2.53e+02	1.58e+01

Table 81. Interleaved Distribution 1-Node Execution Times

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.10e-02	6.67e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	8.33e+00	2.82e-02
Primordial Phase	7.81e+01	1.68e-01
Convert Data Structure	8.03e-01	1.05e-01
Distribution Dependent Time	8.72e+01	2.21e-01
Juxtapositional Phase	8.12e+01	9.34e+00
Other	2.20e-03	4.22e-04
Total	1.68e+02	9.31e+00

Table 82. Interleaved Distribution 2-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.84e-01	5.74e-02
Create Building Blocks	1.00e+00	0.00e+00
Initialize Population	1.98e+00	1.71e-03
Primordial Phase	1.99e+00	5.37e-03
Convert Data Structure	7.83e-01	9.36e-02
Distribution Dependent Time	1.98e+00	5.22e-03
Juxtapositional Phase	9.88e-01	1.15e-01
Other	2.45e+00	7.98e-01
Total	1.50e+00	6.20e-02

Table 83. Interleaved Distribution 2-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.20e-03	4.22e-04
Initialize Population	4.26e+00	1.36e-02
Primordial Phase	3.93e+01	1.67e-01
Convert Data Structure	1.43e+00	1.60e-01
Distribution Dependent Time	4.50e+01	2.07e-01
Juxtapositional Phase	8.27e+01	5.75e+00
Other	2.30e-03	4.83e-04
Total	1.27e+02	5.76e+00

Table 84. Interleaved Distribution 4-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.65e-01	4.51e-02
Create Building Blocks	9.00e-01	2.11e-01
Initialize Population	3.87e+00	3.84e-03
Primordial Phase	3.95e+00	1.93e-02
Convert Data Structure	4.38e-01	4.91e-02
Distribution Dependent Time	3.83e+00	2.20e-02
Juxtapositional Phase	9.75e-01	1.94e-01
Other	2.37e+00	7.97e-01
Total	1.99e+00	1.44e-01

Table 85. Interleaved Distribution 4-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.30e-03	4.83e-04
Initialize Population	2.22e+00	6.84e-03
Primordial Phase	1.96e+01	1.25e-01
Convert Data Structure	1.56e+00	1.69e-01
Distribution Dependent Time	2.34e+01	2.83e-01
Juxtapositional Phase	8.69e+01	7.97e+00
Other	2.70e-03	4.83e-04
Total	1.10e+02	8.02e+00

Table 86. Interleaved Distribution 8-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.65e-01	4.51e-02
Create Building Blocks	8.50e-01	2.42e-01
Initialize Population	7.41e+00	8.20e-03
Primordial Phase	7.92e+00	5.40e-02
Convert Data Structure	4.00e-01	4.05e-02
Distribution Dependent Time	7.37e+00	8.75e-02
Juxtapositional Phase	9.30e-01	1.92e-01
Other	2.05e+00	7.94e-01
Total	2.31e+00	2.06e-01

Table 87. Interleaved Distribution 8-Node Speedups

B.4 Block Distribution.

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.10e-02	6.67e-04
Create Building Blocks	1.10e-03	3.16e-04
Initialize Population	1.64e+01	5.59e-02
Primordial Phase	1.55e+02	5.96e-01
Convert Data Structure	6.20e-01	6.75e-04
Distribution Dependent Time	1.72e+02	6.22e-01
Juxtapositional Phase	8.03e+01	1.58e+01
Other	5.80e-03	1.40e-03
Total	2.53e+02	1.58e+01

Table 88. Block Distribution 1-Node Execution Times

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.11e-02	5.68e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	8.30e+00	2.72e-02
Primordial Phase	6.55e+01	2.23e-01
Convert Data Structure	4.70e+00	4.11e-01
Distribution Dependent Time	7.85e+01	5.43e-01
Juxtapositional Phase	8.06e+01	8.66e+00
Other	2.20e-03	4.22e-04
Total	1.55e+02	8.69e+00

Table 89. Block Distribution 2-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.92e-01	5.36e-02
Create Building Blocks	1.10e+00	3.16e-01
Initialize Population	1.98e+00	4.68e-04
Primordial Phase	2.37e+00	1.01e-02
Convert Data Structure	1.33e-01	1.07e-02
Distribution Dependent Time	2.20e+00	1.72e-02
Juxtapositional Phase	9.96e-01	1.31e-01
Other	2.67e+00	5.44e-01
Total	1.63e+00	7.87e-02

Table 90. Block Distribution 2-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.12e-02	4.22e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	4.24e+00	1.34e-02
Primordial Phase	2.89e+01	1.41e-01
Convert Data Structure	8.51e+00	2.47e-01
Distribution Dependent Time	4.16e+01	3.60e-01
Juxtapositional Phase	8.39e+01	6.61e+00
Other	2.10e-03	3.16e-04
Total	1.18e+02	6.68e+00

Table 91. Block Distribution 4-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.82e-01	3.83e-02
Create Building Blocks	1.10e+00	3.16e-01
Initialize Population	3.87e+00	1.64e-03
Primordial Phase	5.38e+00	3.49e-02
Convert Data Structure	7.29e-02	2.16e-03
Distribution Dependent Time	4.14e+00	4.30e-02
Juxtapositional Phase	9.60e-01	1.79e-01
Other	2.82e+00	7.95e-01
Total	2.14e+00	1.53e-01

Table 92. Block Distribution 4-Node Speedups

Operation/Phase	Execution Time	
	Mean	Standard Deviation
Generate Competitive Template	1.13e-02	4.83e-04
Create Building Blocks	1.00e-03	2.29e-19
Initialize Population	2.20e+00	6.72e-03
Primordial Phase	1.20e+01	8.84e-02
Convert Data Structure	5.89e+00	1.52e-01
Distribution Dependent Time	2.01e+01	2.22e-01
Juxtapositional Phase	9.05e+01	1.06e+01
Other	2.70e-03	4.83e-04
Total	1.06e+02	1.07e+01

Table 93. Block Distribution 8-Node Execution Times

Operation/Phase	Speedup	
	Mean	Standard Deviation
Generate Competitive Template	9.73e-01	4.27e-02
Create Building Blocks	1.10e+00	3.16e-01
Initialize Population	7.46e+00	3.52e-03
Primordial Phase	1.29e+01	1.03e-01
Convert Data Structure	1.05e-01	2.66e-03
Distribution Dependent Time	8.56e+00	1.06e-01
Juxtapositional Phase	8.92e-01	1.54e-01
Other	2.22e+00	6.62e-01
Total	2.40e+00	2.18e-01

Table 94. Block Distribution 8-Node Speedups

Appendix C. *Premature Convergence Experimental Data.*

This appendix contains the raw data from the parallel GA communication strategy performance experiments, including

- execution time as a function of population size for each selection strategy and each sharing strategy for large and small populations (Section C.1), and
- convergence statistics as a function of population size for each communication strategy for large and small populations (Section C.2).

C.1 Execution Time Results.

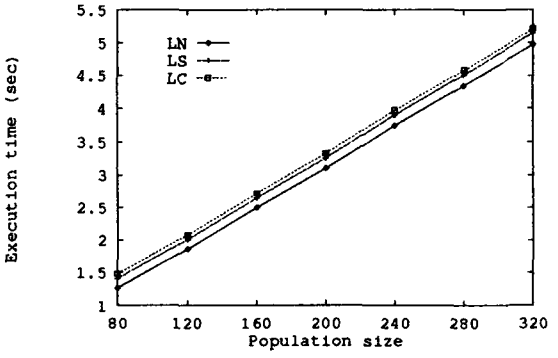

Figure 49. Execution Time – Local Selection Strategies, Small Populations

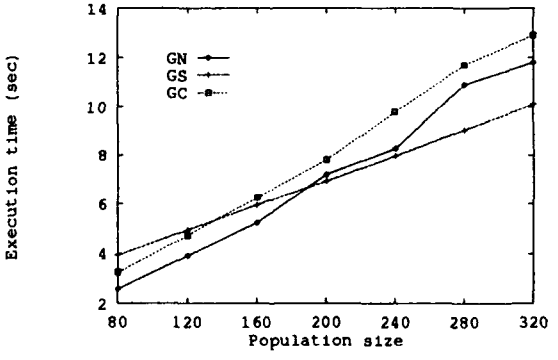

Figure 50. Execution Time – Global Selection Strategies, Small Populations

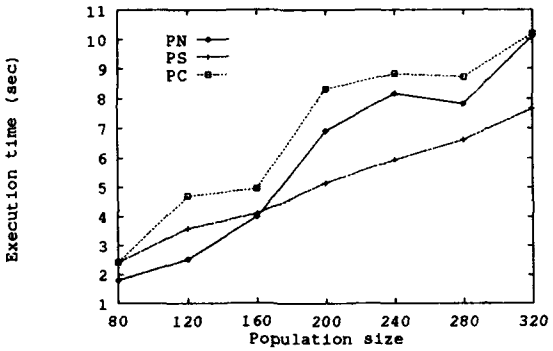

Figure 51. Execution Time – Parallel Selection Strategies, Small Populations

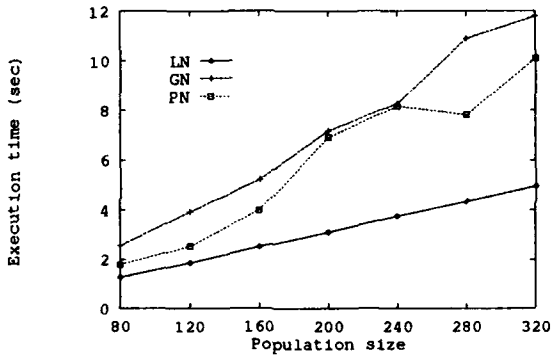

Figure 52. Execution Time – No Sharing Strategies, Small Populations

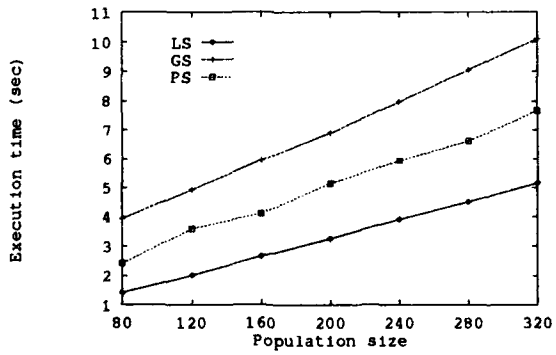

Figure 53. Execution Time – Sharing Strategies, Small Populations

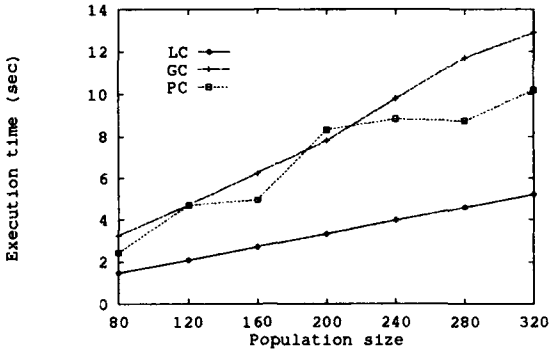

Figure 54. Execution Time – Conditional Sharing Strategies, Small Populations

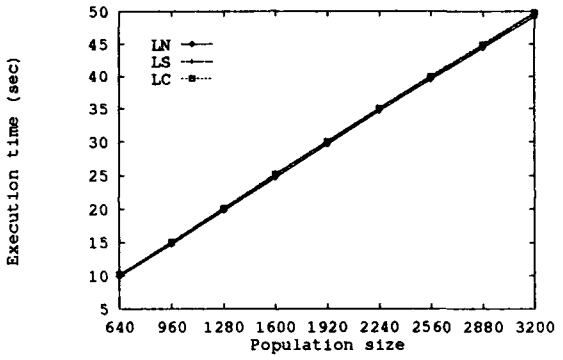

Figure 55. Execution Time – Local Selection Strategies, Large Populations

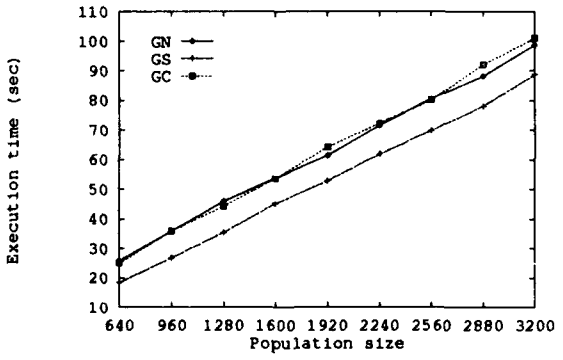

Figure 56. Execution Time – Global Selection Strategies, Large Populations

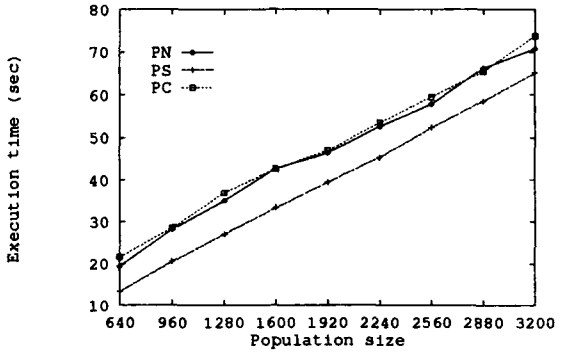

Figure 57. Execution Time – Parallel Selection Strategies, Large Populations

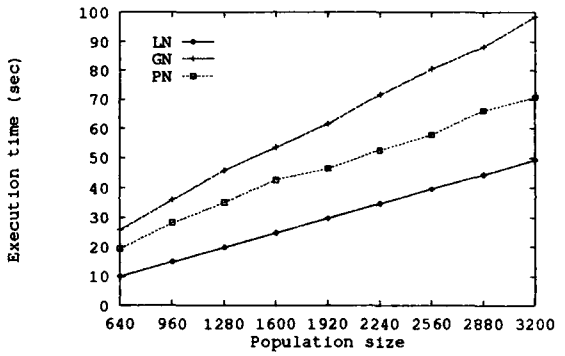

Figure 58. Execution Time – No Sharing Strategies, Large Populations

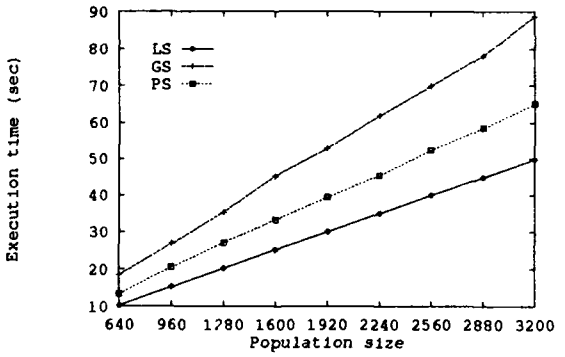

Figure 59. Execution Time – Sharing Strategies, Large Populations

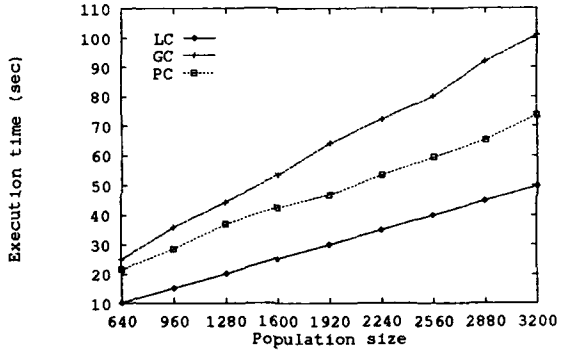

Figure 60. Execution Time – Conditional Sharing Strategies, Large Populations

C.2 Convergence Statistics.

Population Size	Best Efficiency Statistic
80	0.175475
120	0.179334
160	0.034127
200	0.275756
240	0.417403
280	0.335744
320	0.375141

Table 95. Convergence – LN Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.735421
960	0.690678
1280	0.818774
1600	0.837974
1920	0.773545
2240	0.778448
2560	0.759530
2880	0.836116
3200	0.855363

Table 96. Convergence – LN Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.000055
120	0.005281
160	0.000000
200	0.006654
240	0.000932
280	0.102384
320	0.032443

Table 97. Convergence – LS Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.270970
960	0.165808
1280	0.343118
1600	0.448728
1920	0.211212
2240	0.987014
2560	0.831621
2880	0.945857
3200	0.986169

Table 98. Convergence – LS Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.176258
120	0.204836
160	0.368535
200	0.338833
240	0.361256
280	0.389871
320	0.492369

Table 99. Convergence – LC Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.641501
960	0.788789
1280	0.751156
1600	0.755642
1920	0.780675
2240	0.875257
2560	0.821995
2880	0.766922
3200	0.832032

Table 100. Convergence – LC Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.000000
120	0.000000
160	0.086943
200	0.232566
240	0.330757
280	0.389171
320	0.417160

Table 101. Convergence – GN Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.600676
960	0.668768
1280	0.768950
1600	0.689089
1920	0.774709
2240	0.715834
2560	0.767101
2880	0.595963
3200	0.776001

Table 102. Convergence – GN Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.003117
120	0.000000
160	0.001752
200	0.001740
240	0.031254
280	0.228590
320	0.115155

Table 103. Convergence – GS Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.403861
960	0.152572
1280	0.380230
1600	0.213462
1920	0.891523
2240	0.675793
2560	0.880311
2880	0.494462
3200	0.725887

Table 104. Convergence – GS Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.009799
120	0.024899
160	0.299083
200	0.221736
240	0.442417
280	0.524813
320	0.580191

Table 105. Convergence – GC Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.788158
960	0.711784
1280	0.857189
1600	0.694918
1920	0.762510
2240	0.834600
2560	0.886936
2880	0.875530
3200	0.827665

Table 106. Convergence – GC Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.002448
120	0.000000
160	0.020023
200	0.211456
240	0.242646
280	0.333400
320	0.430564

Table 107. Convergence – PN Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.505046
960	0.492894
1280	0.291565
1600	0.462527
1920	0.635694
2240	0.489370
2560	0.536974
2880	0.557232
3200	0.658160

Table 108. Convergence – PN Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.000688
120	0.000563
160	0.000570
200	0.000030
240	0.002229
280	0.018840
320	0.003151

Table 109. Convergence – PS Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.209377
960	0.173146
1280	0.328435
1600	0.483307
1920	0.238900
2240	0.395858
2560	0.253727
2880	0.262649
3200	0.244536

Table 110. Convergence – PS Strategy, Large Population Sizes

Population Size	Best Efficiency Statistic
80	0.002297
120	0.217904
160	0.133283
200	0.139163
240	0.183101
280	0.249313
320	0.395569

Table 111. Convergence – PC Strategy, Small Population Sizes

Population Size	Best Efficiency Statistic
640	0.554424
960	0.570333
1280	0.504046
1600	0.581797
1920	0.537362
2240	0.675596
2560	0.710481
2880	0.686433
3200	0.572514

Table 112. Convergence – PC Strategy, Large Population Sizes

Bibliography

1. Allen, Arnold O. *Probability, Statistics, and Queueing Theory: With Computer Science Applications*. Computer Science and Scientific Computing, San Diego, California: Academic Press, Inc., 1990.
2. Bagley, J. D. *The Behavior of Adaptive Systems which Employ Genetic and Correlation Algorithms*. PhD dissertation, University of Michigan, 1967.
3. Baker, James E. "Reducing bias and inefficiency in the selection algorithm." *Genetic Algorithms and Their Applications: Proceedings of the 2nd International Conference*. 14–21. Hillsdale NJ: Lawrence Erlbaum Associates, 1987.
4. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice* (First Edition). Englewood Cliffs NJ: Prentice Hall, 1988.
5. Brindle, A. *Balancing Diversity and Convergence in Genetic Search*. PhD dissertation, University of Alberta, Alberta, 1981.
6. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Reading MA: Addison-Wesley Publishing Company, 1989.
7. Cohoon, J. P. and others. "A multi-population genetic algorithm for solving the k-partition problem on hypercubes." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 244–248. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
8. Cohoon, J.P. and others. "Punctuated equilibria: a parallel genetic algorithm." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 148–154. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.
9. Davis, Lawrence. "Job Shop Scheduling with Genetic Algorithms." *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. 136–140. 1985.
10. Deb, Kalyonmoy. *A Note on the String Growth in Messy Genetic Algorithms*. Technical Report TCGA Report No. 90006, Tuscaloosa, AL 35487-2908: Department of Engineering Mechanics, The University of Alabama, June 1990.
11. Deb, Kalyonmoy. *Binary and Floating Point Optimization Using Messy Genetic Algorithms*. PhD dissertation, Department of Engineering Mechanics, The University of Alabama, Tuscaloosa, AL 35487-2908, April 1991.
12. DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware*. Englewood Cliffs, NJ 07632: Prentice Hall, Inc., 1989.
13. DeJong, Kenneth A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.*. PhD dissertation, The University of Michigan, Ann Arbor MI, 1975.
14. Dymek, Capt Andrew. *An Examination of Hypercube Implementations of Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92-M, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1992 (AD-A248092).

15. Eshelman, Larry J. and others. "Biases in a crossover landscape." *Proceedings of the Third International Conference on Genetic Algorithms*. 10–19. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1989.

16. Ferguson, David M. and Peter A. Kollman. "Can the Leonard-Jones 6-12 Function Replace the 10-12 Form in Molecular Mechanics Calculations?," *Journal of Computational Chemistry*, *12*(5):620–626 (1991).

17. Frantz, D. R. *Non-linearities in Genetic Adaptive Search*. PhD dissertation, University of Michigan, 1972.

18. Garey, M.R. and D.S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

19. Goldberg, David E. *Optimal Initial Population Size for Binary-Coded Genetic Algorithms*. Technical Report, Tuscloosa AL: University of Alabama, 1985.

20. Goldberg, David E. "Genetic Algorithms and Walsh Polynomials: Part I, A Gentle Introduction," *Complex Systems*, *3*:129–152 (1989).

21. Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison–Wesley Publishing Company, 1989.

22. Goldberg, David E. "Sizing Populations for Serial and Parallel Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

23. Goldberg, David E. and others. "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex Systems*, *3*:493–530 (1989).

24. Goldberg, David E. and others. "Messy Genetic Algorithms Revisited," *Complex Systems*, *4*:415–444 (1990).

25. Goldberg, David E. and others. "Don't Worry, Be Messy." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 24–30. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

26. Goldberg, David E. and Robert Lingle. "Alleles, loci, and the traveling salesman problem." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. 154–159. Hillsdale NJ: Lawrence Erlbaum Associates, 1988.

27. Grefenstette, John J. *A User's Guide to Genesis*. Technical Report, Nashville TN: Vanderbilt University, 1986.

28. Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor MI: The University of Michigan Press, 1975.

29. Humphrey, Watts S. *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1990.

30. Jog, Prasana and Dirk Van Gucht. "Parallelisation of probabilistic sequential search algorithms." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 170–176. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

31. Kargupta, Hillol, et al. *Ordering Genetic Algorithms and Deception.* Technical Report IlliGAL Report No. 92006, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801: Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign, April 1992.

32. Kelley, Al and Ira Pohl. *A Book on C: Programming in C.* The Benjamin/Cummings series in computing and information science, Redwood City, CA 94065: The Benjamin/Cummings Publishing Company, Inc., 1990.

33. King, Jonathan. "Deciphering the Rules of Protein Folding," *C & E News*, 32–54 (April 10, 1989).

34. Kommu, Venkantarama. Personal communication at 21st International Conference on Parallel Processing, August 1992.

35. Kommu, Venkantarama. *Enhanced Genetic Algorithms in Constrained Search Spaces with Emphasis on Parallel Environments.* PhD dissertation, Department of ECE, University of Iowa, Iowa City, Iowa, 1992. in preparation.

36. Kommu, Venkantarama and Irith Pomeranz. "Effective Communication in a Genetic Algorithm." *Proceedings of the 21st International Conference on Parallel Processing III, Algorithms & Applications*, edited by Quentin F. Stout. 310–317. 1992.

37. Kosak, Corey and others. "A Parallel Genetic Algorithm for Network-Diagram Layout." *Proceedings of the Fourth International Conference on Genetic Algorithms.* 458–465. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

38. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing.* Englewood Cliffs, NJ: Prentice Hall, 1992.

39. Lybrand, Terry P. "Computer Simulation of Biomolecular Systems Using Molecular Dynamics and Free Energy Perturbation Methods." *Reviews in Computational Chemistry* edited by Kenny B. Lipkowitz and Donald B. Boyd, chapter 8, 295–320, VCH Publishers, Inc, 1990.

40. Muhlenbein, H. "Parallel Genetic Algorithms, Population Genetics, and Combinatorial Optimization." *Proceedings of the Third International Conference on Genetic Algorithms.* 416–421. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

41. Muhlenbein, H. and others. "The Parallel Genetic Algorithm as Function Optimizer." *Proceedings of the Fourth International Conference on Genetic Algorithms.* 271-278. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

42. Nayeem, Akbar, et al. "A Comparative Study of the Simulated-Annealing and Monte Carlo-with-Minimization Approaches to the Minimum-Energy Structures of Polypeptides: [Met]-Enkephalin," *Journal of Computational Chemistry*, *12*(5):594–605 (1991).

43. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design.* Yourdon Press Computing Series, Englewood Cliffs, New Jersey 07632: Prentice Hall, Inc., 1988.

44. Pearl, Judea. *Heuristics.* Reading MA: Addison–Wesley Publishing Company, 1989.

45. Pettey, Chrisila B. and others. "A parallel genetic algorithm." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 155–161. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

46. Pettey, Chrisila B. and Michael R. Leuze. "A theoretical investigation of a parallel genetic algorithm." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

47. Rich, Elaine and Kevin Knight. *Artificial Intelligence* (2nd Edition). New York: McGraw Hill, 1983.

48. Ripoll, Daniel R., et al. "On the Multiple-Minima Problem in the Conformational Analysis of Polypeptides V. Application of the Self-Consistent Electrostatic Field and the Electrostatically Driven Monte Carlo Methods to Bovine Pancreatic Trypsin Inhibitor," *PROTEINS: Structure, Function, and Genetics*, *10*:188–198 (1991).

49. Robertson, George G. "Parallel implementation of genetic algorithms in a classifier system." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 148–154. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

50. Rumbaugh, James. *Object-oriented Modeling and Design*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1991.

51. Sannier, Adrian V. and Erik D. Goodman. "Genetic learning procedures in distributed environments." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 148–154. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

52. Sawyer, George A. *Functional optimization using parallel genetic algorithms*. Technical Report, Wright-Patterson AFB OH: Air Force Institute of Technology, 1989.

53. Sawyer, George Allen. *Extraction and Measurement of Multi-Level Parallelism in Production Systems*. MS thesis, AFIT/GCE/ENG/90D-04, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230498).

54. Sikora, R. *Analysis of Deception for Permutation Problems*. Unpublished manuscript, 1991.

55. Smith, D. "Bin Packing with Adaptive Search." *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. 202–206. 1985.

56. Sobell, Mark G. *A Practical Guide to the UNIX System*. The Benjamin/Cummings Series in Computing and Information Science, Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1989.

57. Spiessens, Piet and Bernard Manderick. "A Massively Parallel Genetic Algorithm: Implementation and First Analysis." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 279–285. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

58. Tanese, Reiko. "Parallel genetic algorithms for a hypercube." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 170–176. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

59. von Freyberg, Berthold and Werner Braun. "Efficient Search for All Low Energy Conformations of Polypeptides by Monte Carlo Methods," *Journal of Computational Chemistry*, *12*(9):1065–1076 (1991).

60. Yourdon, Edward. *Modern Structured Analysis*. Yourdon Press Computing Series, Englewood Cliffs, New Jersey 07632: Yourdon Press, 1989.

Vita

Captain Laurence D. Merkle earned his bachelor's degree in Computers and Systems Engineering from the Rensselaer Polytechnic Institute in 1987. He earned his commission through Air Force ROTC. Upon entering active duty in 1988, he was assigned to Air Force Logistics Command's Artificial Intelligence Program Management Office. While there, he taught introductory AI classes and trained engineers and other AFLC personnel to develop small rule-based expert systems. He left AFLC in 1991 to attend AFIT.

Permanent address: 225 Balmoral Dr
Kettering, OH 45429

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms

5. FUNDING NUMBERS

6. AUTHOR(S)
Laurence D. Merkle, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/92D-08

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Wright Laboratories (AFMC)
Materials Directorate
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Genetic algorithms (GA) are highly parallelizable, robust semi-optimization algorithms of polynomial complexity. The most commonly implemented GAs are "simple" GAs (SGAs). Reproduction, crossover, and mutation operate on solution populations. Deceptive and GA-hard problems are provably difficult for simple GAs. Messy GAs (MGA) are designed to overcome these limitations.

The MGA is generalized to solve permutation type optimization problems. Its performance is compared to another MGA's, an SGA's, and a permutation SGA's. Against a fully deceptive problem the generalized MGA (GMGA) consistently performs better than the simple GA. Against an NP-complete permutation problem, the GMGA performs better than the other GAs. Against DeJong function f2, the GMGA performs better than the other MGA, but not as well as the SGA.

Four parallel MGA data distribution strategies are compared and not found to significantly affect solution quality. The interleaved strategy obtains near linear speedup. The indexed, modified indexed, and block strategies obtain "super-linear speedup," indicating that the sequential algorithm can be improved.

Population partitioning impacts implementation of selection and crossover. Experiments which compare the solution quality, execution time, and convergence characteristics of three selection algorithms and three solution sharing strategies are performed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Processing, Genetic Algorithms, Optimization	186
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL